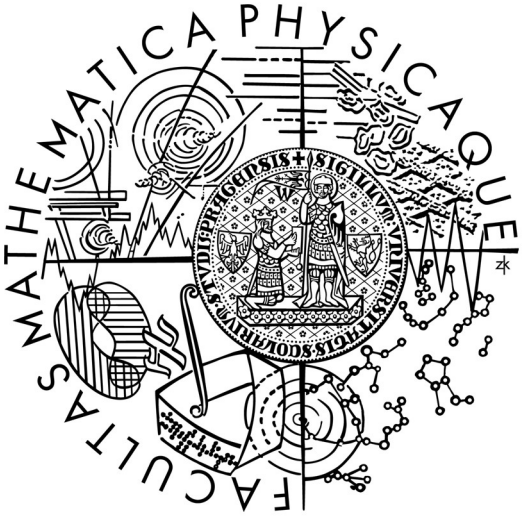
Univerzita Karlova v Praze
Matematicko-fyzikalní fakulta

BAKALAŘSKÁ PRÁCE



David Matoušek

Nástroj pro kontrolu systémových volání pro Windows

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Yaghob, Ph.D.

Studijní program: Informatika, programování

2007

Rád bych poděkoval mému vedoucímu bakalářské práce za pomoc při jejím zpracování. Dále bych rád poděkoval autorům použité literatury a všem výzkumníkům, kteří se dělí o své poznatky v oblasti, se kterou souvisí má práce. V neposlední řadě bych rád poděkoval kolegům, kteří přislíbili spolupráci na dalším vývoji software, který je základem této práce.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 14. dubna 2007 David Matoušek

Název práce: Nástroj pro kontrolu systémových volání pro Windows

Autor: David Matoušek

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Yaghob, Ph.D., Katedra softwarového inženýrství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze, Malostranské náměstí 25, Praha 1

e-mail vedoucího: Jakub.Yaghob@mff.cuni.cz

Abstrakt: Tato práce se zabývá návrhem a implementací software, který umožní svým uživatelům detailně monitorovat a případně i pozměňovat chování procesů na operačních systémech Microsoft Windows 2000, XP a Server 2003. V předkládané podobě je software rozdělen do dvou hlavních částí. Systémová část se stará o interakci s jádrem operačního systému, ve kterém zachycuje systémové volání, jejichž další obsluhu pak nechává na externích modulech, pro které definuje jednoduché API. Nezbytnou součástí práce je tak i hlavní externí modul, který rozebírá jednotlivá volání a utváří detailní záznam o chování zvolených procesů. Samotný text práce pak obsahuje širší pojednání o problematice návrhu a implementace takového software, nabízí srovnání s jinými srovnatelnými nástroji a zmiňuje budoucí možnosti rozvoje tohoto díla. Součástí práce je i CD se zdrojovými kódy a funkční kompilací software. Software využívá část GNU C Library.

Klíčová slova: systémové volání, hákování funkcí, monitorování chování

Title: A tool for checking systems calls on Windows

Author: David Matoušek

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D., Department of Software Engineering, Charles University, Malostranské náměstí 25, Prague

Supervisor's e-mail address: Jakub.Yaghob@mff.cuni.cz

Abstract: This thesis identifies issues of process monitoring on Windows NT operating systems family and describes our implementation of an infrastructure for process monitoring. The system comprises of a core component and user defined external modules. The kernel component hooks system functions and redirects calls to the external modules running in the user space. The core component dispatches the calls to user supplied external modules that take care of the call processing. The core component defines simple API for the external modules that does not require deep knowledge of kernel programming. Moreover, this architecture simplifies versioning of the external modules. It makes them independent of a particular kernel build as long as the Windows kernel API they are monitoring is preserved. The thesis also compares our work with similar existing software and outlines future directions of development. Source codes and an executable compilation of the software are available on the attached CD. The software uses a part of GNU C Library.

Keywords: system calls, function hooking, behaviour tracking

Obsah

1 Úvod

Narozdíl od nedávné minulosti se v dnešním světě informačních technologií setkáváme s prvkem informační bezpečnosti téměr ve všech aplikacích dostupných běžnému uživateli. S rozvinutím Internetu získala počítačová bezpečnost nový rozměr. Běžně se na Internetu setkáme s hrozbami jako jsou spam, phishing, spyware, počítačoví červi apod. Přirozeně tak vzniklo odvětví počítačového průmyslu, ve kterém se jednotlivci i firmy zabývají analýzou a monitorováním bezpečnostních incidentů, tvorbou software i hardware pro omezení rizik a dopadů bezpečnostních incidentů jako jsou například anti-viry, firewally, anti-spamové filtry, verifikátory obsahu apod. Velkého rozvoje dosáhl výzkum v oblasti ochrany citlivých dat, šifrování a digitální autentifikace.

Tato práce spadá do oboru informační bezpečnosti a věnuje se analýze chování software na operačních systémech Windows. Základem práce je software (dále jen *Software*), který umožňuje sledovat interakci mezi operačním systémem a procesy na daném systému. Moderní výzkum v oblasti prevence a detekce škodlivých kódů se čím dál více zaměřuje na rozpoznání nebezpečného chování nezávisle na jeho implementaci, zatímco detekce na bázi porovnávání vzorků známých škodlivých kódů je na ústupu. Je však mnoho druhů činností, které lze u programů sledovat. *Software* implementuje sledování na úrovni systémových volání na rozhraní uživatelské části operačního systému a jádra systému. Volba této úrovně bude diskutována později v této práci.

Ač by se zdálo, že využití *Software* je z předchozího jasně dáno a omezeno na zkoumání chování potenciálně nebezpečných aplikací, jsou možnosti využití mnohem širší. Zkoumat chování programů totiž nemá smysl pouze u škodlivých kódů, ale lze také zkoumat chování bezpečnostního software jako jsou anti-viry nebo dnes populární osobní firewally a tzv. HIPS (Host Intrusion Prevention Systems). Zde lze pomocí analýzy chování zjišťovat možné bezpečnostní nedostatky nebo testovat dodržení původního návrhu takového software. Pro další rozšíření využití *Software* byla navíc do jeho návrhu zahrnuta možnost sledované chování jistým způsobem ovlivňovat pomocí externích modulů, které lze za běhu přidávat a odebírat. Kromě samotné analýzy chování procesů, tak lze cíleně měnit jejich chování bez nutnosti zasahovat do jejich binární podoby.

Cílem této práce je vytvořit nástroj, který zachycuje systémová volání a dovoluje jejich další zpracování na uživatelské úrovni včetně možnosti modifikace chování aplikace pomocí kontroly volání.

Kapitola 2 této práce se věnuje úvodu do problematiky. Jsou v ní definovány základní pojmy důležité pro porozumění dalšímu obsahu práce. Také jsou zde podrobněji diskutovány cíle práce a požadavky na implementační část práce. V části 2.2 je pak dán výčet uvažovaných technik, které se nabízejí pro řešení stanovených cílů a ze kterých pak v části 2.3 provádíme výběr jedné techniky, která je použita pro implementaci základu *Software*. Kapitola 3 se věnuje technickým detailům návrhu a implementace tohoto díla a osvětluje důležité mechanismy práce *Software*. V kapitole 4 se čtenář seznámí se základním ovládáním produktu a s prací s externím modulem pro sledování chování procesů nazvaným SystemSpy. Kapitola 5 srovnává relevantní vlastnosti implementovaného řešení s konkurenčními produkty. Závěrečná kapitola 6 pak shrnuje výsledky práce a především nabízí možnosti dalšího rozvoje výsledků této práce.

Práce obsahuje dvě textové přílohy. Příloha A je kompletní a detailní dokumentace aplikačního rozhraní pro tvorbu externích modulů a obsahuje i odkazy na příklady použití dokumentovaných funkcí.

Součástí práce je i přiložené médium, které obsahuje elektronickou verzi této práce, zdrojové kódy *Software* i jeho binární spustitelnou kompilaci a další pomocné soubory. Struktura souborů na přiloženém médiu je popsána v příloze B této práce.

2 Základní definice a návrh

V této kapitole si představíme cíle projektu a základní možnosti řešení. Také zde definujeme některé pojmy, které jsou nezbytné pro porozumění dalšímu textu jako např. *hákování*, *hákovaná funkce*, *hákovaný proces*, *hákované vlákno*, *handler*, *code flow*, *system-wide hook*, *hook chain*, *SSDT*, *nativní funkce, aktuální obsluha, původní obsluha*.

2.1 Požadavky na sledování a cíle projektu

Vzhledem ke své složitosti je přirozené, že operační systémy Windows s jádrem NT nabízejí více než jednu možnost, jak sledovat chování procesů, které na nich běží. Při návrhu sledovacího systému je proto nezbytné stanovit si požadavky, které musí náš systém splňovat, a cíle, kterých chceme pomoci sledování dosáhnout.

Jeden ze základních požadavků byl již zmíněn v úvodu. Sledovací systém má sloužit pro analýzu malware. U kódů tohoto typu lze očekávat, že jejich autor implementoval nějaký druh ochrany proti sledování jako je například ochrana proti debugování, obfuskace kódu, vykonávání kódu na nízké úrovni operačního systému, šifrování kódu a dat, samomodifikující se kód, modifikace chování operačního systému apod. Bylo by dobré mít možnost sledovat chování i takto chráněných kódů a v případech, kdy toho dosáhnout nelze, získat alespoň informaci o tom, že sledovaný kód je nějak chráněn.

Dalším požadavkem je pokrytí co největšího počtu aktivit sledovaného procesu, ale zároveň vyvarovat se sledování aktivit irelevantních vzhledem k záměrům sledování. Ve většině případů nemá smysl zabývat se analýzou každé instrukce programu a pokud někdy taková situace nastane, existuje již řada specializovaných programů jako jsou disassemblery, debuggery, dekompilátory apod., které lze pro tento účel použít.

Některé techniky, které umožňují sledování chování procesu v systému, nutně ovlivňují všechny procesy systému i v případě, že takovou techniku chceme využít ke sledování jediného procesu. Navíc libovolná implementace sledování má jistý dopad na výkonnost systému nebo alespoň na výkonnost sledovaných procesů. Vzniká tak požadavek, aby výkonostní dopad byl co nejmenší a především, aby byl minimalizován dopad na ty procesy systému, které sledovat nechceme.

Monitorovací software lze naprogramovat jednolitě. Tím se omezí veškeré jeho možnosti na pouhé sledování, ale na druhé straně tím lze získat na výkonnosti. Jinou možností je návrh, který by podporoval rozšiřitelnost bez zásahů do kódu samotného sledovacího systému. Již na začátku se uvažovalo o budoucnosti projektu, tedy o možnostech rozšíření výsledné funkcionality nad rámec této práce. Proto byl zvolen druhý model tak, aby *Software* podporoval externí moduly, které lze dotvářet samostatně bez nutnosti modifikací hotového jádra produktu.

2.2 Možnosti implementace sledování

Na základě výše zmíněných požadavků lze uvažovat o následujících technikách, které lze

využít pro účely sledování. U každé techniky jsou uvedeny výhody a nevýhody, které měly dopad na výslednou volbu pro návrh řešení. Hodnotným zdrojem informací v této oblasti je např. [2].

Nejprve však objasníme pojem *hákování* (angl. hooking), který se zde objevuje velmi často. Obecně je *hákování* technika, při které se kód existující funkce nebo ukazatel na funkci nahradí kódem nebo ukazatelem, který přesměruje vykonávání kódu na jiný kód. Při nahrazování kódu se tak typicky děje na prvních instrukcích funkce. Funkci, která je takto změněna, nazýváme *hákovanou funkcí* (angl. hooked function), proces, který je takto ovlivněn nazýváme *hákovaný proces* (angl. hooked process). Podobně pak vlákno hákovaného procesu, jehož činnost vede na volání hákované funkce, nazýváme *hákované vlákno* (angl. *hooked thread*). Při volání hákované funkce získává nová obsluha (*hook handler* nebo jen *handler*) všechny parametry původní funkce a může nad nimi konat libovolné operace. Dále je technicky možné zavolat originální kód hákované funkce a ve většině případů získat zpět řízení po návratu z originálního kódu. V tomto bodě lze sledovat a modifikovat výstupní hodnoty originálního kódu. Hovoříme zde o tzv. změně *toku kódu* nebo angl. o změně *execution flow* nebo *code flow*.

Hákování knihovních funkcí v user mode

Tato technika využívá skutečnosti, že každý proces v systému má k dispozici vlastní adresový prostor, do kterého jsou mu načteny kopie systémových i aplikačních knihoven, které program využívá. Program má ve své vlastní struktuře v paměti seznamy knihovních funkcí, které využívá, a operační systém při zavádění programu tyto tabulky vyplní platnými ukazately na požadované funkce v jeho adresovém prostoru tak, že dohledá příslušné funkce v kopiích knihoven. Ukazatele na funkce, které program tzv. importuje, jsou umístěny ve struktuře s názvem IAT (Import Address Table).

Nabízí se tedy minimálně dvě možnosti, jak pozměnit code flow. V okamžiku načítání programového modulu po vyplnění ukazatelů v IAT můžeme tyto ukazatele přepsat. V okamžiku, kdy proces volá knihovní funkci, nahlíží do IAT, aby zjistil adresu funkce ve svém adresovém prostoru, a vykonávání kódu pokračuje na zjištěné adrese. Pokud jsme adresu funkce v IAT modifikovali, získává řízení kód na námi zvolené adrese.

Druhou možností je přímá změna instrukcí na adrese knihovní funkce. IAT tedy obsahuje originální adresu, ale kód, který se na ní nachází je nepůvodní. Typicky se namísto prvních instrukcí vkládá instrukce skoku, která přesměruje code flow na námi zvolenou adresu. Alternativa k instrukci skoku je využití jednobytové instrukce breakpoint. Sledující aplikace se tak chová jako debugger a získává řízení od systému v okamžiku, kdy dojde k vykonání této instrukce.

V obou těchto případech je nutné, abychom na adrese, kam přesměrováváme code flow, měli vlastní kód, který zajistí požadovanou funkcionalitu. K tomu je nezbytné alokovat v hákovaném procesu paměť a naplnit ji příslušným kódem. Nejjednodušší implementace této techniky načte do cílového procesu dynamicky linkovanou knihovnu a tím přenechá starosti alokací a relokací operačnímu systému.

Výhody techniky

- Zdaleka nejširší možné pokrytí aktivit procesu. Lze monitorovat libovolné chování, které se zakládá na libovolné knihovní funkci, což obsahuje prakticky vše, kromě interní práce s pamětí procesu.

- Dokonalá izolace. Prakticky jsou ovlivněny pouze ty procesy, které chceme sledovat, a tyto procesy jsou ovlivněny pouze v případech, kdy volají sledované funkce.
- Lze implementovat i jako čistě user mode řešení. Programování ovladačů běžících v kernel mode operačního systému je náročné, čistě user mode řešení je tak obecně snažší na implementaci.
- Dostupnost dokumentace. Většina důležitých knihovních funkcí, které lze využívat v user mode, je plně dokumentovaných výrobcem systému. Není tak problém s interpretací jednotlivých volání.

Nevýhody techniky

- Nelze zamezit vyhýbání se sledování. Sledovaný proces není nikdy nucen využít knihovních funkcí v user mode. Vždy lze implementovat vlastní kód se stejnou funkcionalitou, jehož vykonávání nelze touto technikou podchytit.
- Možné kolize. Tato technika je velmi často používána jiným softwarem. Je obtížné implementovat tuto techniku tak, aby byla kompatibilní s ostatním software při snaze každého takového softwaru hákovat stejnou funkci. I v případě perfektní implementace na naší straně nelze ovlivnit implementaci cizího software, která může být nekvalitní. Výsledkem kolize bývá pád hákovaného procesu nebo dokonce pád celého systému.
- U čistě user mode řešení je obtížné implementovat sledování všech procesů v systému (angl. *system-wide hook*). Navíc při sledování většího počtu procesů zároveň mohou být celkové nároky nezanedbatelné vzhledem k nutnosti přítomnosti kopie vlastního kódu v každém sledovaném procesu.
- Řada funkcí systémových knihoven je implementována ve více variantách, pokud chceme podchytit chování v dané oblasti, je nutné sledovat všechny takové funkce. Například mnoho funkcí je implementováno pro podporu řetězců s velikostí znaku jeden byte, což usnadňuje programování malých aplikací, a zároveň i pro podporu s velikostí dva byty na znak, což je nativní pro jádro operačního systému. Názorným příkladem jsou API funkce pro práci se službami systému **EnumServicesStatusA**, **EnumServiceGroupW**, **EnumServicesStatusExA** a **EnumServicesStatusExW**, které se svojí funkcionalitou překrývají.

Hákování knihovních funkcí v kernel mode

Podobně jako v případě předchozí techniky se lze chovat i v kernel mode. Je zde však několik důležitých rozdílů. Nejpodstatnější odlišností je existence jen jedné kopie kódu jádra systému. Libovolná změna v kernelu tak nutně ovlivňuje všechny procesy systému. V kernelu kromě kódu samotného operačního systému také existují ovladače, které získávají řízení na základě své role v systému. Například ovladač souborového systému získává řízení při požadavku procesu na otevření souboru, který se v daném souborovém systému nachází. Ovladače jsou svoji strukturou velmi podobné programovým modulům v user mode a tak i zde existují tabulky pro import knihovních funkcí, které lze alternovat. A také, podobně jako v user mode, lze modifikovat počáteční instrukce funkcí samotného kernelu.

Výhody techniky

- Stále velmi široké pokrytí aktivit procesu, ikdyž menší než u předchozí techniky. Lze monitorovat libovolné chování procesu, které nějakým způsobem interaguje se

systémem. Lze monitorovat veškerou komunikaci mezi procesem a systémem, přístup k systémovým strukturám, komunikaci s ostatními procesy, komunikaci s hardware apod.

- Nelze se vyhnout sledování. Například pokud chce proces pracovat se souborem na disku, musí vždy projít přes volání v jádře. Jediná možnost, jak se vyhnout sledování, je zavedení vlastního ovladače a útok na sledovací systém nebo jeho obcházení. To však lze vždy detekovat. Navíc je pro sledovaný proces obtížné detekovat, že je sledován.

Nevýhody techniky

- Nutně a vždy ovlivňuje celý systém. Je nutná dodatečná implementace pro rozlišení volání procesů, které sledovat chceme a těch, které nás nezajímají.
- Nedostupnost dokumentace. Pro většinu důležitých funkcí v kernel mode není veřejně dostupná řádná dokumentace od výrobce systému. Je nutné se spolehnout na vlastní výzkum nebo na práci nezávislých odborníků.
- Nutnost překonat ochranu paměti v kernelu a vyřešit obtíže synchronizace při změně kódu v kernelu. Operační systémy s jádrem NT se snaží chránit před náhodnými změnami důležitých struktur v kernelu. Je třeba znát tyto mechanismy a obejít je, aby při změně paměti nedošlo k pádu systému. Navíc je třeba dbát na synchronizaci a to především u strojů s více procesory, které je nutné u netriviálních změn v kernelu pečlivě synchronizovat. Nedostatky v implementaci samotného hákování se mohou projevit nahodilými a těžko analyzovatelnými pády systému. Novější verze systémů Windows (Windows XP a vyšší, zvláště pak Windows Vista a vyšší) navíc implementují dodatečné ochrany, které mají zcela zamezit změnám v kernelu.
- Pro opravdu široké pokrytí aktivit procesu je nutné hákovat funkce v kernelu, samotné hákování funkcí, které používají ovladače nestačí. Řádná implementace takového hákování je obtížná a podobně jako v user mode hrozí kolize s jiným software, ikdyž tato technika není příliš užívaná.

Hákování System Service Descriptor Table

Jádro operačního systému Windows poskytuje některé systémové služby user mode procesům prostřednictvím systémové knihovny `ntdll.dll`. Tato knihovna exportuje tzv. *nativní funkce*, které přecházejí při svém vykonávání do kernel mode. V kernelu existuje struktura nazvaná System Service Descriptor Table (*SSDT*), která obsahuje ukazatele na všechny služby, které jádro systému exportuje. Při volání nativní funkce z `ntdll.dll` dochází k přepnutí z user mode do kernel mode. Obsluha v kernel mode získá při tomto přepnutí index služby, která se má v kernelu vykonat. Pomocí tohoto indexu se v SSDT vyhledá ukazatel na funkci, která zajišťuje požadovanou službu a předá se jí řízení. Ukazatele na funkce služeb lze přepsat a získat tak řízení v případě, že se vyvolá nativní funkce. Podobně jako pro služby jádra existuje v systému ještě tabulka pro služby grafického interface. Struktura, ve které se nachází tabulka adres funkcí, které obsluhují tyto služby, se nazývá System Service Descriptor Table Shadow (*SSDT Shadow*). Definujme si zde ještě pojmy *aktuální obsluha* (nebo *aktuální funkce*) a *původní obsluha* (nebo *původní funkce*). Při závádění hákování nazveme jako aktuální takovou obsluhu, na kterou ukazuje záznam SSDT v okamžiku jeho nahrazování. Jako původní obsluhu nazveme takovou funkci, která náleží operačnímu systému, tedy takovou, na kterou ukazuje záznam v SSDT předtím, než došlo k hákování SSDT libovolným software.

Výhody techniky

- Poměrně široké pokrytí aktivit procesu, opět ale menší než u předchozí techniky. Monitorovat lze pouze chování, které vede na volání nativních funkcí. To však obsahuje veškeré chování, při kterém dochází k nějaké práci procesu s okolím. Lze tedy monitorovat meziprocesovou komunikaci, práci se soubory, s registry, s procesy a vlákny, použití synchronizace. V případě funkcí tabulky SSDT Shadow lze monitorovat i práci s grafickým interface, kam částečně spadá i obsluha vstupních zařízení jako je klávesnice nebo myš.
- Tato technika vykazuje stejné vlastnosti co do možnosti detekce a vyhýbání se sledování jako hákování funkcí v kernel mode.
- Oproti předchozí technice je implementace hákování v SSDT výrazně jednodušší a potřebná synchronizace je triviální.
- Na rozdíl od předchozí techniky je především díky [3] dostupná poměrně kvalitní dokumentace k funkcím v SSDT, přestože výrobce systému poskytuje dokumentaci pouze k malému množství nativních funkcí.
- Implicitní možnost několikanásobného hákování bez problémů kolize. Typická implementace hákování SSDT nahrazuje ukazatele zvolených služeb ukazately na vlastními funkce, ve kterých se po případném zpracování předá řízení na adresu funkce, která byla v SSDT před samotným hákováním. Většinou se nezjišťuje, zda šlo o adresu originální systémové obsluhy nebo o adresu do ovladače jiného software, který implementuje SSDT hákování. Přirozeně tak vzniká řetězec hákování (angl. *hook chain*), ve kterém po zpracování obsluhy našeho hákování získává řízení kód, jenž hákoval bezprostředně před námi. Navíc případná odinstalace hákování nenaruší hook chain ostatních aplikací.

Nevýhody techniky

- Podobně jako v případě předchozí techniky, i tato nutně a vždy ovlivňuje celý systém a je nutná dodatečná implementace pro rozlišení volání procesů, které sledovat chceme a těch, které nás nezajímají.
- Stejně jako u hákování funkcí v kernel mode i zde se mění obsah struktur jádra, je zde nutnost překonávat ochranu operačního systému.
- Dostupná dokumentace je pro novější systémy Windows (Windows XP a vyšší) neúplná a nepřesná. Navíc např. [3] se věnuje výhradně systémovým službám a nikoliv službám grafického interface, tedy funkcím v SSDT Shadow.

Hákování Interrupt Descriptor Table

Každý procesor, který tvoří základ počítače s instalovaným operačním systémem Windows, implementuje tzv. přerušení (angl. *interrupts*). Přerušení nastává v okamžiku nějaké speciální události a znamená pozastavení výkonu kódu na procesoru, kde k přerušení došlo, a předání řízení obsluze přerušení (angl. interrupt service routine). Například přerušení 0 nastává při pokusu o vykonání instrukce dělení nulou, přerušení 1 slouží k pozastavení procesoru po výkonu každé instrukce a používá se při debugování, přerušení 3 se rovněž používá k debugování jako tzv. breakpoint. Vyšší přerušení jsou definovatelná operačním systémem. Pro sledování běhu programu jsou zajímavá například přerušení stisku klávesy nebo přerušení systémové služby. U techniky hákování SSDT byl zmíněn přechod z user mode do kernel mode při volání nativní funkce. Právě tento přechod je na starších systémech NT

implementován jako přerušení. Novější systémy však z výkonostních důvodů používájí specializovanou instrukci k dosažení téhož. Více o přerušeních lze nalézt např. v [5].

Procesor disponuje speciálním registrem zvaným Interrupt Descriptor Table Register. Tento registr obsahuje ukazatel v jádře na tabulku Interrupt Descriptor Table (*IDT*), ve které jsou uloženy záznamy o obsluhách přerušení procesoru. A právě zde lze proces obsluhy přerušení hákovat.

Výhody techniky

- Lze podchytit speciální aktivity procesu, které se jinak obtížně monitorují. Na starších systémech (Windows 2000 a starší) lze hákovat volání systémových služeb, čímž se dosáhne rozsahu pokrytí jako u metody hákování SSDT.
- Implementace na velmi nízké úrovni operačního systému. Prakticky nehrozí kolize s cizím software.

Nevýhody techniky

- Na novějších systémech se nepoužívá přerušení pro volání systémových služeb. Rozsah pokrytí je tak u nových systémů (Windows XP a vyšší) značně omezen.
- Hook na této úrovni ovlivňuje celý systém a je zde tedy nutnost dodatečné implementace pro rozlišení procesů.
- Na rozdíl od klasických hákovacích technik je hákování IDT specifické tím, že po případném volání originální obsluhy nezískává volající řízení po návratu zpět. Je tedy možné ovlivňovat nebo sledovat volání pouze na vstupu. To je pro řadu aplikací velmi zásadní nedostatek a prakticky diskvalifikuje tuto techniku pro naše účely.
- Každý procesor má vlastní tabulku přerušení. Na multiprocesorových strojích je nutné změnit záznamy IDT na více místech a přitom dbát na řádnou synchronizaci.

Použití vrstevnatého modelu ovladačů a filtrace

Vrstevnatý model ovladačů v jádře Windows umožňuje zapojit vlastní ovladač do řetězce ovladačů, které zpracovávají určitý požadavek. Typickým příkladem je systém souborů. Požadavek na práci se souborem putuje přes několik ovladačů, z nichž každý má svoji specifickou funkci. Chceme-li do systému přidat prvek šifrování souborů, můžeme implementovat ovladač a umístit jej nad všechny ostatní ovladače. V takovém případě můžeme data zapisovaná do souborů modifikovat dříve než doputují k ovladačům souborového systému. Na disku tak adresářová struktura souborového systému zůstane zachována a pouze data každého souboru budou šifrována. Pokud však ovladač implementujeme jinak a umístíme jej mezi ovladač, který pracuje se samotným diskem, a ovladač, který zapisuje strukturu souborového systému, budou šifrována nejen samotná data, ale rovněž i adresářová struktura. Při správné implementaci bude šifrování pro ostatní ovladače transparentní. Namísto šifrovacího ovladače můžeme samozřejmě implementovat pouze ovladač monitorovací, který nezmění žádná data požadavku.

Výhody techniky

- Vrstevnatý model ovladačů a filtrování je plně podporováno a dokumentováno výrobcem. Oproti hákování má tedy tu výhodu, že není třeba překonávat ochrany operačního systému. Implementace samotného zavedení se tak omezuje na použití

funkcí, za které ručí výrobce systému, a tím se tato technika stává mnohem stabilnější a spolehlivější, než libovolná hákovací technika.

- Přirozená možnost více filtrů různých software.
- Je obtížné vyhnout se sledování, lze tak učinit pouze přímým útokem na sledovací ovladač v kernelu.

Nevýhody techniky

- Šířka pokrytí je poměrně malá. Lze sledovat práci se soubory, u novějších systémů i práci s registry, filtrování podporují i např. ovladače klávesnice. Nelze však sledovat meziprocesovou komunikaci, práci s procesy a vlákny nebo synchronizací.
- Řešení na této úrovni ovlivňuje celý systém a je zde tedy nutnost dodatečné implementace pro rozlišení procesů.

2.3 Výběr techniky pro implementaci sledování

V předcházející části jsme rozebrali pět možných technik, které lze použít pro účely sledování chování procesů na systémech Windows s jádrem NT. Již při popisu nevýhod jsme vyloučili hákování IDT vzhledem k malému rozsahu pokrytí na nových systémech a především nemožnosti získat řízení na výstupu hákované funkce. Rovněž z důvodu malého pokrytí lze vyloučit použití vrstevnatého modelu a filtrace. Všechny zbývající metody jsou pro implementaci sledování vhodné a každá má řadu svých specifických výhod i nevýhod. Hákování v user mode lze obejít, proto tato technika není příliš vhodná vzhledem k nasazení proti malware. Při rozhodování mezi zbylými dvěma technikami, hákování funkcí v kernel mode a hákování SSDT, volíme mezi širším rozsahem pokrytí aktivit procesu a značně jednodušší implementací. Vzhledem k tomu, že hákování SSDT nabízí stále velmi dobré pokrytí, a vzhledem k relativně dobré dokumentaci, byla tato technika zvolena jako základní kámen při implementaci *Software*.

3 Realizace návrhu

Software se skládá ze čtyř částí – ovladače, systémové služby, klienta pro zavádění pravidel a externích modulů. Z pohledu implementace lze *Software* rozdělit na jádro systému, které obsahuje ovladač, systémovou službu a klienta, a na externí moduly. V této kapitole jsou popsány role jednotlivých částí a také jejich vzájemná provázanost a komunikace. V závěru kapitoly je uvedeno schéma znázorňující průběh zachycení jednoho volání hákované funkce.

3.1 Ovladač

Vzhledem ke zvolené technice hákování SSDT je nezbytné implementovat ovladač, jelikož SSDT je součástí jádra operačního systému a je nepřístupná pro aplikace z user mode. Výjimku tvoří možnost přístupu přes mapování fyzické paměti, ale tato možnost byla zcela odstraněna na novějších systémech Windows (Windows Server 2003 SP1 a vyšší). Pro funkci, kterou požadujeme hákovat, modifikuje ovladač příslušný ukazatel v SSDT tak, aby přesměroval code flow do své vlastní tabulky, kterou nazýváme *Generic SSDT Handlers*. Zde se na zásobník uloží identifikace funkce a dále se pomocí instrukce skoku předá řízení handleru jednotnému pro všechny hákované funkce, který se jmenuje *Generic SSDT Stub*. Tento handler zjistí, zda volání pochází z procesu a vlákna, které chceme sledovat, a v kladném případě informuje službu o události volání hákované funkce. V kontextu hákovaného procesu pak ovladač již jen plní požadavky služby, mezi které patří práce s pamětí hákovaného procesu nebo požadavek na volání aktuální obsluhy.

Filtrace volání procesů, které sledovat nechceme, není teoreticky na této úrovni nezbytná. Pokud bychom však odsunuli tuto filtraci na úroveň služby, znamenalo by to výrazné snížení výkonu celého systému. Volání hákované funkce libovolným procesem v systému by totiž muselo použít komunikační kanál mezi ovladačem a službou, což vyžaduje nákladné operace na synchronizaci přístupu k sdíleným datům. Je tedy výhodné implementovat filtraci co nejdříve po zajištění samotného volání. Ovladač si proto udržuje bitovou mapu, nazývanou též *rychlý SSDT filtr*, pro každý proces a funkci SSDT, aby mohl ihned rozlišit, která volání předat ke zpracování službě a kterými se nezabývat. Tato bitová mapa je udržováná pomocí zpráv ze služby, která spravuje samotný systém pravidel v user mode.

Ovladač má i další využití. Při svém zavedení registruje tzv. notifikační rutiny, které získávájí řízení v případě vzniku nebo zániku procesů nebo vláken v systému a také v případě načítání modulů do již existujících procesů. Na základě těchto událostí si ovladač udržuje přehled o běžících procesech a vláknech, který distribuuje také službě. Starší verze Windows (Windows 2000 a starší) plně nepodporují odregistrování notifikačních rutin. Z tohoto důvodu není možné ovladač ze systému odebrat, jakmile jednou dojde k jeho instalaci. To znamená, že pro kompletní odinstalování *Software* je nutný restart systému.

3.2 Služba

Existence služby je důležitá především vzhledem k možnostem dalšího vývoje *Software.* V předkládané verzi celý systém vyžaduje administrátorská práva jak při instalaci,

tak při samotném běhu aplikace. Takové omezení příliš nevadí v aktuální situaci, kdy je využití aplikace omezeno na sledování chování, protože existuje pouze jeden prakticky použitelný externí modul. S budoucím rozvojem by však toto omezení bylo závažným nedostatkem. Oddělení klienta od služby může napomoci v implementaci podpory užívání *Software* s účty neprivilegovaných uživatelů.

Hlavním úkolem služby je správa a řízení všech událostí. Služba spravuje systém pravidel a poskytuje rozhraní pro připojení klienta, který zpracovává uživatelské požadavky na modifikaci pravidel. Dále služba instaluje ovladač a řídí jeho činnost. Mezi službou a ovladačem existují dva komunikační kanály. První kanál se využívá pro základní řízení celého systému a pro synchronizaci a správu přehledů o běžících procesech a vláknech systému. Tento kanál je využíván i pro úpravy filtrovacích bitových map na základě požadavků, které uživatel předkládá ve formě pravidel.

Druhý kanál se používá při volání hákované funkce. V takovém případě jako první získává řízení ovladač a zasílá tímto kanálem informaci o dané události službě. Služba reaguje na tuto událost předáním řízení handleru v externím modulu, který je přiřazen k hákované funkci.

Služba pak slouží k odbavování některých požadavků z handlerů směrem k ovladači nebo k dalším handlerům. Pokud například handler žádá o čtení paměti cizího procesu, předá služba požadavek ovladači, který danou operaci vykoná a informuje službu o výsledku popř. předá požadovaná data.

Veškerá komunikace mezi službou a ovladačem ohledně požadavků vztahujících se k obsluze hákované funkce se děje přes tento druhý kanál. Tento kanál se tedy využívá i pro transfer dat při čtení a zápisech z a do paměti hákovaného procesu. Při pozdějších testech se zjistilo, že implementace tohoto kanálu pomocí jediného bufferu sdílené paměti mezi službou a ovladačem je výkonostně velmi omezující a prakticky znemožnuje reálné nasazení *Software* pro obsluhu system-wide hook. V takovém případě je totiž odezva systému značně zpomalena. Při aplikaci *Software* k monitorování chování nějakého software tento jev příliš nevadí, jelikož ani v případě nutnosti použít system-wide hook nebude mít ve většině případů zpomalení systému vliv na chování zkoumaných procesů. Avšak pro možné budoucí moduly, u kterých by se dalo uvažovat o nasazení na počítačích běžných uživatelů, je odezva systému kritická vlastnost. Jedno z důležitých vylepšení v dalším vývoji *Software*, tak musí být zmnožení bufferů, které jsou použity pro komunikaci v druhém kanálu.

3.3 Uživatelský klient

Uživatelský klient je součástí binárního souboru, ve kterém je i služba. Při spuštění uživatelem se program chová jako klient, zatímco když jej spouští Service Control Manager, chová se jako služba. Uživatel vždy spouští klienta s parametrem jména souboru s pravidly, které chce aplikovat. Klient pravidla načte a pokud jsou syntakticky v pořádku, bude se snažit tato pravidla předat službě. Klient si nejprve ověří, zda je služba instalována a spuštěna. Pokud tomu tak není, pokusí se o službu nainstalovat. Dále se napojí na službou zřízené rozhraní, kam zasílá daná pravidla. Služba klientovi odpovídá informacemi o výsledcích zpracování pravidel a klient tyto informace zprostředkovává uživateli. Toto jsou jediné funkce klienta.

Detailní informace o pravidlech a formátu souboru s pravidly jsou popsány v kapitole 4.

3.4 Externí moduly

Jakmile dojde k volání hákované funkce procesem, jehož aktivity nás zajímají, ovladač pošle službě informaci o volání a ta jej přesměruje dále do externího modulu. Uživatelem definovaná pravidla pro každou hákovanou funkci přesně specifikují, který modul a která jeho funkce slouží k obsluze volání dané hákované funkce. Služba implementuje pro každou hákovanou funkci její vlastní hook chain v rámci *Software*, takže je možné pro jednu hákovanou funkci definovat více než jeden handler najednou. Výsledné chování je stejné, jako kdyby každý handler sám implementoval SSDT hákování právě v tom pořadí, ve kterém jsou uvedeny v souboru pravidel. To znamená, že handler posledního pravidla bude první, který získá od služby řízení.

Externí moduly jsou standardní dynamicky linkované knihovny, které exportují své handlery. Jeden modul může obsahovat i více handlerů, ale zároveň může používat jeden handler pro obsluhu více hákovaných funkcí.

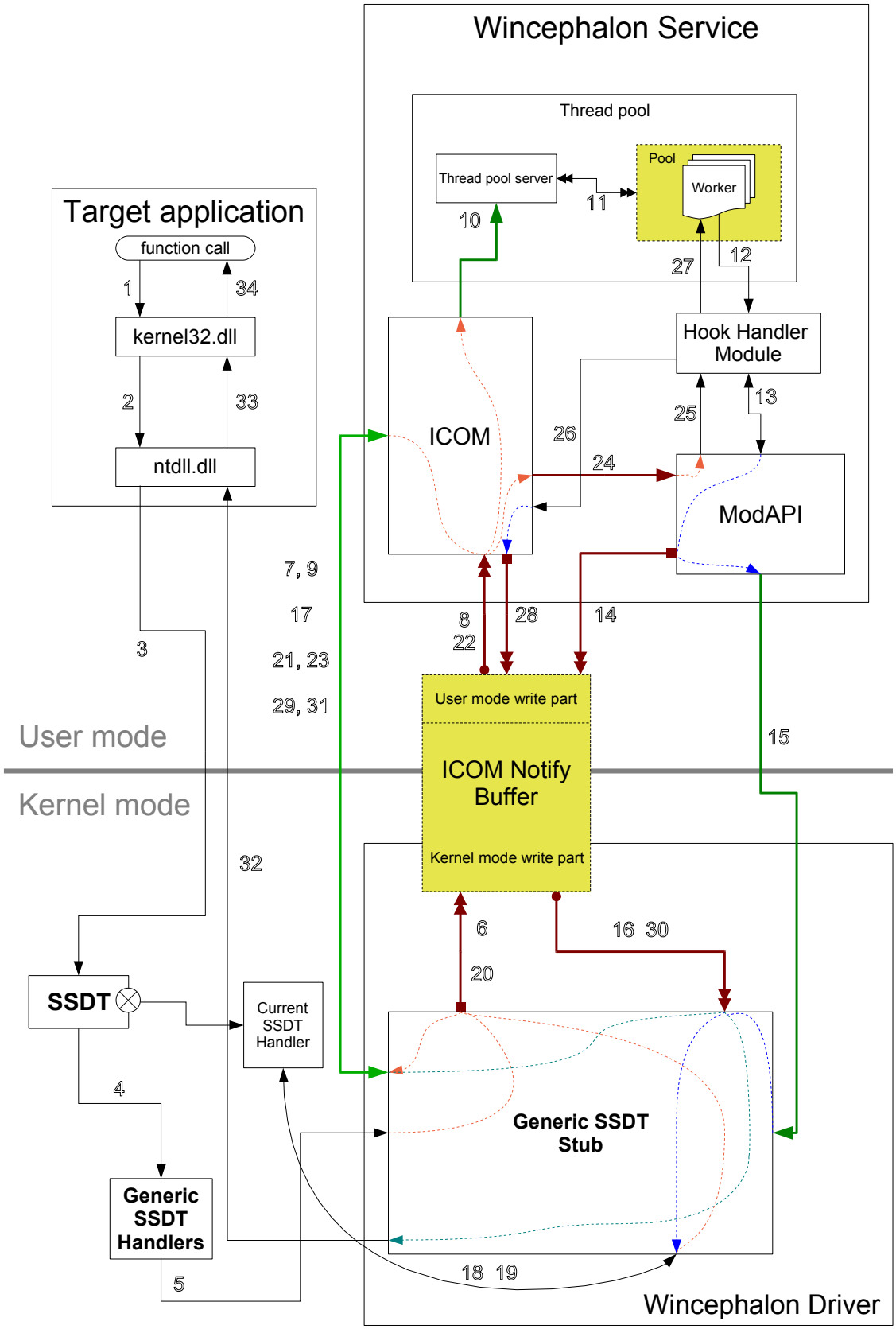
Pro tvorbu externích modulů je definováno jednoduché API, které nazýváme *ModAPI*. Při používání těchto funkcí je nutné si uvědomit, že všechny funkce handleru, na které vede odkaz přes hákovací pravidlo, mohou být obecně vykonávány paralelně ve více vláknech. Je proto nezbytné řešit synchronizaci ke všem nelokálním zdrojům, které tyto funkce používají. Detailní popis ModAPI externích modulů je náplní přílohy A této práce.

Jedna z funkcí ModAPI, **WcpCallNextHandler**, slouží k předání řízení dalšímu handleru v hook chain. Jestliže tedy první handler, který byl instalován posledním pravidlem pro nějakou hákovanou funkci, zavolá **WcpCallNextHandler**, předá služba řízení druhému handleru v pořadí, tedy tomu, k jehož instalaci přispělo předposlední pravidlo k dané funkci. V okamžiku, kdy handler na nějaké úrovni dokončí svoji činnost, navrátí se řízení handleru vyšší úrovně. Přirozeně je možné implementovat handler, který handler nižší úrovně nezavolá. V takovém případě se handlery na nižších úrovních o volání hákované funkce vůbec nedoví. Handler nejnižší úrovně, tedy ten, který je v hook chain na posledním místě, může také zavolat funkci **WcpCallNextHandler**, ale protože žádný nižší handler již neexistuje, interpretuje služba toto volání jako požadavek na volání aktuální obsluhy hákované funkce, který předá ovladači k vykonání.

V prezentované verzi je k dispozici jeden hlavní modul nazvaný SystemSpy a dále několik menších ukázkových modulů, které demonstrují použití jednotlivých API funkcí na krátkých příkladech.

3.5 Schéma volání

Situace, která nastává v případě, že uživatelská aplikace zavolá funkci, která vede na volání hákované nativní funkce, je značně netriviální proces. Schéma tohoto procesu je zobrazeno na obr. 1 na následující straně (pozn. *Wincephalon* je kódový název projektu).



Obr. 1: Schéma znázorňující průběh zachycení volání hákované funkce

1. Uživatelská aplikace volá funkci Windows API, např. `CreateFileA` z `kernel32.dll`.

2. Toto volání pokračuje do `ntdll.dll`, např. funkcí `NtCreateFile`.

3. Nativní `Nt*` funkce předávájí řízení do kernelu pomocí `int 2E` nebo `sysenter`.

4. Volání pokračuje v kernelu na adrese, která je definovaná v SSDT. Pokud je volaná funkce hákována, je adresa v SSDT změněna tak, aby ukazovala do tabulky v nestránkované paměti, kterou nazýváme *Generic SSDT Handlers*.

5. Záznam v tabulce *Generic SSDT Handlers* je jednoduchý skok do ovladače na adresu funkce *Generic SSDT Stub* (*GSS*). V případě, že volající aplikace není sledována, což se zjistí na rychlém SSDT filtru, pokračuje vykonávání bodem 18 a 19 a následně body 32 až 34.

6. Funkce *GSS* používá paměť alokovanou v kernelu, která je namapovaná do uživatelského procesu služby, k přenosu informací o sledovaném volání. Tuto sdílenou pamět nazýváme *ICOM Notify Buffer*. Tato paměť je rozdělena na dvě části. První část je určena k zapisování z ovladače a ke čtení ze služby, druhá část je určena k zapisování ze služby a čtení z ovladače.

7. Když jsou data připravena v bufferu, *GSS* pošle signál vláknu modulu *ICOM* ve službě. Toto vlákno nazýváme *ICOM Notify Thread*.

8. *ICOM* přečte data z *ICOM Notify Buffer* a vytvoří si lokální kopii, aby mohl být buffer znovu použit v co nejkratším čase.

9. *ICOM* pak signalizuje ovladači, že buffer je možné použít pro další požadavky.

10. *ICOM Notify Thread* pošle zprávu serveru *Thread pool*, že je zde nový požadavek ke zpracování.

11. *Thread pool server* najde volný *Worker thread* a přiřadí mu požadavek ke zpracování.

12. *Worker thread* zavolá řídící funkci *Hook Handler Module*, která je asociována k volané `Nt*` funkci. Tato funkce je umístěna v externí DLL knihovně a může vykonat libovolný kód.

13. Navíc však tento kód může používat tzv. *ModAPI* funkce, které importuje z knihovny `wcpmodapi.dll`. Tyto API funkce slouží např. k získání infomací o sledovaném volání a také o procesu, který sledované volání vykonal. Některé z těchto funkcí okamžitě vrátí výsledek.

14. Jiné funkce *ModAPI* jsou složitější a k jejich dokončení je nutné komunikovat s ovladačem. Mezi takové funkce patří i **WcpCallNextHandler**, která slouží k zavolání dalšího handleru, který je asociován k sledované funkci. K jedné sledováné funkci může být asociováno i více handlerů z různých modulů. V momentě, kdy poslední v handlerů zavolá **WcpCallNextHandler**, zapíše se zpráva

do *ICOM Notify Buffer*

15. a pošle se signál ovladači, aby zprávu zpracoval.

16. Ovladač si zkopíruje příchozí zprávu
17. a signálem informuje *ICOM* o tom, že uživatelská *ICOM Notify Buffer* bufferu je uvolněna pro další požadavek.

18. V našem příkladu přichází zpráva z **WcpCallNextHandler**. Ovladač proto předá řízení aktuálnímu handleru sledované funkce (*Current SSDT Handler*) v kontextu volajícího procesu.

19. Jakmile funkce skončí,
20. ovladač zkopíruje její návratovou hodnotu do *ICOM Notify Buffer*
21. a pošle signál do *ICOM* o nové zprávě.

22. Jako obvykle si *ICOM* zkopíruje data do lokální paměti
23. a pošle signál zpět do ovladače, takže ovladač může znovu zapisovat do své části sdílené paměti.

24. *ICOM Notify Thread* přepošle tuto informaci vláknu, které zavolalo **WcpCallNextHandler**.

25. Tím dojde k dokončení této *ModAPI* funkce a navrácení do posledního *Hook Handler Module* s návratovou hodnotou z *Current SSDT Handler*. Funkce v *Hook Handler Module*, která takto dostane zpět řízení může opět vykonat libovolný kód.

26. Když tato funkce skončí, řízení dostane předchozí handler. Když takto skončí funkce prvního handleru, řízení se dostane do modulu *ICOM*.

27. Hned na to je *Worker thread* vráceno do *Thread pool* a je možné jej použít pro další požadavek.

28. *ICOM*, který získal návratovou hodnotu z prvního handleru, ji zapíše do *ICOM Notify Buffer*
29. a odešle signál ovladači.

30. Zpráva je zkopírována do lokální paměti
31. a pomocí signálu z ovladače do *ICOM* se *ICOM Notify Buffer* uvolní pro další požadavky.

32. Řízení a získaná návratová hodnota pak putují zpět do `ntdll.dll`,
33. dále do `kernel32.dll`
34. a nakonec dostane řízení zpět volající uživatelská aplikace.

4 Uživatelské rozhraní, definice pravidel

Uživatel spouští *Software* příkazem `Wincephalon.exe` vždy s jedním parametrem, který specifikuje soubor s pravidly, popř. se jedná o parametr `-uninstall`, který se používá k odinstalování programu. Při prvním spuštění programu se souborem pravidel dojde k nenápadné instalaci jádra programu. To obnáší instalaci systémové služby a ovladače, jejich zavedení, spuštění a inicializaci. Požadavek na odinstalování programu je pak implementován jako zastavení služby a odebrání služby i ovladače. Při zastavování služby automaticky dochází také k odebrání všech definovaných pravidel.

4.1 Formát souboru pravidel

V souboru s pravidly je každý řádek zpracováván samostatně. Prázdné řádky a řádky, jejichž první znak je '#', jsou při zpracování ignorovány. Uživatel definuje pravidla v následujícím formátu:

1. Pravidlo pro hákování funkce se píše ve tvaru

   `add SSDT_HOOK as <HOOK_NUMBER> name <FUNC_NAME1> flags <FLAG_LIST> module <MODULE_NAME> function <FUNC_NAME2>`

   kde `<FUNC_NAME1>`, `<FUNC_NAME2>` a `<MODULE_NAME>` jsou řetězce psané v uvozovkách, `<HOOK_NUMBER>` je kladný integer a `<FLAG_LIST>` je výčet možných vlastností tohoto hákování oddělených čárkou bez vložených mezer. Číslo `<HOOK_NUMBER>` musí být pro každé hákování unikátní. `<FLAG_LIST>` může obsahovat hodnoty `HOOK_USER_CALLS`, `HOOK_KERNEL_CALLS` a `HOOK_CHILDREN`. `HOOK_USER_CALLS` znamená, že hákování bude platné pro volání funkce pocházející z user mode; podobně `HOOK_KERNEL_CALLS` znamená, že hákování bude platné pro volání funkce pocházející z kernel mode; `HOOK_CHILDREN` znamená, že pokud je sledován nějaký proces, pak každý jeho potomek je také sledován nezávisle na splnění dalších filtrovacích pravidel. Jméno řídícího modulu `<MODULE_NAME>` je relativní vůči adresáři, ve kterém se nachází `Wincephalon.exe`. `<FUNC_NAME1>` je jméno funkce, kterou chceme hákovat. `<FUNC_NAME2>` je jméno handleru v daném modulu, která získá řízení při volání hákované funkce. Tento handler musí být exportován řídícím modulem a musí mít předepsaný prototyp.

2. Filtrovací pravidla, tedy omezení procesů a vláken daného hákovacího pravidla, se píší ve tvaru

   `add HOOK_RULE for <HOOK_NUMBER> [flags HOOK_EXISTING] type <RULE_TYPE> <RULE_VALUE>`

   kde `<HOOK_NUMBER>` je kladný integer, který odpovídá číslu `<HOOK_NUMBER>` nějakého hákovacího pravidla. K tomuto hákovacímu pravidlu se pak vztahuje filtrovací pravidlo. Volitelně pravidlo obsahuje podřetězec `flags HOOK_EXISTING`. Pokud pravidlo tento podřetězec obsahuje, bude se pravidlo srovnávat i s již existujícími procesy, v opačném případě dojde ke sledování pouze takových procesů a vláken, které pravidlu odpovídají a které vzniknou až po zavedení pravidla. `<RULE_TYPE>` definuje typ pravidla, možnosti jsou tři: `ALL_CIDS` pro aplikaci sledování na všechny

procesy a vlákna, tedy system-wide hook – v tomto případě se `<RULE_VALUE>` neuvádí; `SINGLE_CID` pro aplikaci na jeden konkretní proces nebo vlákno – v tomto případě je `<RULE_VALUE>` číselný identifikátor procesu nebo vlákna; `REGEXP` pro aplikaci na všechny procesy, jejichž cesta odpovídá regulárnímu výrazu v `<RULE_VALUE>`, který se píše v uvozovkách. Knihovna regulárních výrazů je plně převzata z GNU C Library, která používá POSIX syntax. Srovnávání je case insensitive.

3. Pravidlo pro odebrání hákovacího pravidla včetně všech jeho filtrovacích pravidel se píše ve tvaru

   ```
   remove SSDT_HOOK id <HOOK_NUMBER>
   ```

   kde `<HOOK_NUMBER>` je číslo sledovacího pravidla.

4. Pro odebrání všech hákovacích i filtrovacích pravidel lze použít předpis

   ```
   remove SSDT_HOOK all
   ```

5. Pravidlo pro odebrání modulu, který byl využíván jako handler sledování, je tvaru

   ```
   unload MODULE name <MODULE_NAME>
   ```

   kde `<MODULE_NAME>` je jméno modulu zapsané jako řetězec v uvozovkách. Modul nesmí být odebrán, dokud všechna pravidla s ním spojená nebyla odebrána.

4.2 Příklad použití modulu SystemSpy

Pro účely této práce byl implementován sledovací modul `systemspy.wcp`, který exportuje jedinou funkci `GenericSSDTHandler`.

Zde jsou reálné příklady použití pravidel:

1. Vytvoříme soubor `rules.conf` obsahující tyto tři řádky

   ```
   add SSDT_HOOK as 1 name "NtCreateFile" flags HOOK_USER_CALLS,HOOK_CHILDREN module "modules\systemspy.wcp" function "GenericSSDTHandler"
   add HOOK_RULE for 1 flags HOOK_EXISTING type REGEXP "pro"
   add HOOK_RULE for 1 type SINGLE_CID 1000
   ```

   a aplikujeme tato pravidla příkazem

   ```
   Wincephalon.exe rules.conf
   ```

   V tomto příkladu nejprve definujeme hákování funkce **NtCreateFile** pro volání z user mode, která se přesměrují do funkce `GenericSSDTHandler` modulu `systemspy.wcp`, který je uložen v adresáři `modules`. Adresář `modules` se musí nacházet v adresáři, kde je uložen `Wincephalon.exe`. Toto pravidlo dále požaduje, aby byli hákováni i potomci všech hákovaných procesů. Dále se definují dvě filtrovací pravidla pro toto hákovací pravidlo. První filtrovací pravidlo požaduje sledování všech procesů, v jejichž cestě se nachází řetězec "`pro`", jedná se tedy například o všechny procesy spuštěné ze standardní složky "`Program Files`". Zároveň se požaduje, aby byly sledovány již existující procesy, které tomuto pravidlu odpovídají. Poslední řádek definuje pravidlo pro sledování všech v budoucnu vytvořených procesů nebo vláken, kterým bude přidělen identifikátor `1000`.

2. Vytvoříme soubor `rules-uninstall.conf` obsahující tyto dva řádky

```
remove SSDT_HOOK all

unload MODULE name "modules\systemspy.wcp"
```

a aplikujeme tato pravidla příkazem

```
Wincephalon.exe rules-uninstall.conf
```

Aplikací souboru pravidel jsme nejprve docílili odebrání všech pravidel ze systému a následně odebrání modulu s handlery k těmto pravidlům. Odinstalace všech pravidel vztahujících se k modulu, který chceme odebrat, je nezbytná. Pokud bychom měli aktivní pravidla vázaná na modul, který odebíráme, došlo by k pádu systému. Dosáhli jsme tedy zastavení definovaných sledování. Kdykoliv však můžeme začít nová sledování zavedením nových pravidel.

3. Vyvoláním příkazu

```
Wincephalon.exe -uninstall
```

dojde k odinstalaci *Software* ze systému. To znamená ukončení definovaných sledování a odebrání služby i ovladače *Software* ze systému. Pokud bychom chtěli *Software* znovu nainstalovat a použít, je nutné nejprve restartovat systém.

4.3 Ukázka výstupu modulu SystemSpy

Výstup modulu `modules\systemspy.wcp` se zapisuje do souboru `WcpSSDT.txt` v adresáři, ve kterém je umístěn `Wincephalon.exe`. Formát výstupu je jednotný pro všechny funkce.

Zde jsou reálné příklady výstupu:

1. Volání a návrat funkce **NtCreateFile**:

```
[05-03-2007 13:42:43.0077] NtCreateFile(0x6C288, 0xC0100080, 0x6C224, 0x6C25C, 0x0, 0x0
, 0x3, 0x1, 0x40, 0x0, 0x0) called by PID = 936, TID = 908 (process path = "\Device\Har
ddiskVolume1\WINNT\system32\notepad.exe")
  FileHandle = 0x0006C288
  DesiredAccess = 0xC0100080
    ACCESS_MASK = (FILE_READ_ATTRIBUTES | SYNCHRONIZE | GENERIC_WRITE | GENERIC_READ)
  ObjectAttributes = 0x0006C224
    .Length = 0x18
    .RootDirectory = NULL
    .ObjectName = 0x0006C264
      .Length = 0x1E
      .MaximumLength = 0x21A
      .Buffer = 0x000721D0
        *PWSTR = "\??\PIPE\lsarpc"
    .Attributes = 0x40
      ULONG = (OBJ_CASE_INSENSITIVE)
    .SecurityDescriptor = NULL
    .SecurityQualityOfService = 0x0006C248
      .Length = 0xC
      .ImpersonationLevel = 0x2
        SECURITY_IMPERSONATION_LEVEL = SecurityImpersonation
      .ContextTrackingMode = 0x1
        SECURITY_CONTEXT_TRACKING_MODE = TRUE
      .EffectiveOnly = 0x1
        BOOLEAN = TRUE
  IoStatusBlock = 0x0006C25C
  AllocationsSize = NULL
  FileAttributes = 0x0
  ShareAccess = 0x3
    ULONG = (FILE_SHARE_READ | FILE_SHARE_WRITE)
  CreateDisposition = 0x1
    ULONG = FILE_OPEN
  CreateOptions = 0x40
```

```
    ULONG = (FILE_NON_DIRECTORY_FILE)
  EaBuffer = NULL
  EaLength = 0x0

[05-03-2007 13:42:43.0077] NtCreateFile returned 0x0 = STATUS_SUCCESS (called by PID =
936, TID = 908)
  FileHandle = 0x0006C288
    *PHANDLE = 0x0000011C
  IoStatusBlock = 0x0006C25C
    .Status = 0x0
    .Information = 0x1
      ULONG_PTR = FILE_OPENED
```

Každé volání se skládá ze dvou částí. První část výstupu je zapsána před voláním aktuální obsluhy hákované funkce a jsou zde vypsány hodnoty všech parametrů funkce včetně rozebrání struktury vstupních parametrů. Druhá část je zápis po návratu z volání aktuální obsluhy funkce, zde jsou vypsány parametry výstupní.

První řádek výpisu před voláním obsahuje po řadě: datum a čas volání, jméno volané funkce, číselné hodnoty parametrů v hexadecimálním tvaru oddělené čárkou, číslo procesu PID a číslo vlákna TID, ve kterých došlo k volání, a nakonec jméno hlavního souboru volajícího procesu. Dále následuje strukturovaný výpis parametrů funkce. Výpis každého parametru začíná jeho jménem a hodnotou, která je zapsáná ve tvaru, jenž odpovídá jejímu typu. Poté, pokud se jedná o strukturu nebo je možné nějakým způsobem hodnotu dále rozepsat, následují odsazené řádky s detaily o parametru. Například u struktur jsou vypsány po řadě všechny položky této struktury včetně hodnot s tím, že opět může dojít k bližší specifikaci některých položek, která se pozná podle odsazení vyššího stupně. U konstant a výčtů jsou dány řetězcové názvy hodnot, u stringů se zobrazí hodnota v čitelné podobě. Řádky s detaily u hodnot nedělitelných typů začínají vždy jménem typu.

Výstup po návratu funkce obsahuje po řadě: datum a čas návratu z funkce, jméno funkce, návratovou hodnotu v číselné i řetězcové podobě, pokud je tato známa, a informaci o volajícím procesu a vláknu. Dále následuje odsazený výpis výstupních argumentů a jejich hodnot ve stejném formátu jako u výpisu před voláním funkce.

2. Návrat funkce **NtReadFile** a volání funkce **NtWriteFile**:

```
[05-03-2007 13:44:01.0296] NtReadFile returned 0x0 = STATUS_SUCCESS (called by PID =
1028, TID = 872)
  IoStatusBlock = 0x0006B298
    .Status = 0x0
    .Information = 0x10F
  Buffer = 0x01410000
    5B 45 78 74 53 68 65 6C 6C 46 6F 6C 64 65 72 56                  [ExtShellFolderV
    69 65 77 73 5D 0D 0A 44 65 66 61 75 6C 74 3D 7B                  iews]..Default={
    35 39 38 34 46 46 45 30 2D 32 38 44 34 2D 31 31                  5984FFE0-28D4-11
    43 46 2D 41 45 36 36 2D 30 38 30 30 32 42 32 45                  CF-AE66-08002B2E
    31 32 36 32 7D 0D 0A 7B 35 39 38 34 46 46 45 30                  1262}..{5984FFE0
    2D 32 38 44 34 2D 31 31 43 46 2D 41 45 36 36 2D                  -28D4-11CF-AE66-
    30 38 30 30 32 42 32 45 31 32 36 32 7D 3D 7B 35                  08002B2E1262}={5
    39 38 34 46 46 45 30 2D 32 38 44 34 2D 31 31 43                  984FFE0-28D4-11C
    46 2D 41 45 36 36 2D 30 38 30 30 32 42 32 45 31                  F-AE66-08002B2E1
    32 36 32 7D 0D 0A 0D 0A 5B 7B 35 39 38 34 46 46                  262}....[{5984FF
    45 30 2D 32 38 44 34 2D 31 31 43 46 2D 41 45 36                  E0-28D4-11CF-AE6
    36 2D 30 38 30 30 32 42 32 45 31 32 36 32 7D 5D                  6-08002B2E1262}]
    0D 0A 57 65 62 56 69 65 77 54 65 6D 70 6C 61 74                  ..WebViewTemplat
    65 2E 4E 54 35 3D 66 69 6C 65 3A 2F 2F 46 6F 6C                  e.NT5=file://Fol
    64 65 72 2E 68 74 74 0D 0A 0D 0A 5B 2E 53 68 65                  der.htt.....[.She
    6C 6C 43 6C 61 73 73 49 6E 66 6F 5D 0D 0A 43 6F                  llClassInfo]..Co
    6E 66 69 72 6D 46 69 6C 65 4F 70 3D 30 0D 0A                     nfirmFileOp=0..

[05-03-2007 13:44:01.0437] NtWriteFile(0xCC, 0x0, 0x0, 0x0, 0x6FB24, 0x8DB38, 0xB, 0x0
, 0x0) called by PID = 1028, TID = 872 (process path = "\Device\HarddiskVolume1\WINNT\
system32\notepad.exe")
  FileHandle = 0x000000CC
```

```
Event = NULL
ApcRoutine = NULL
ApcContext = NULL
IoStatusBlock = 0x0006FB24
Buffer = 0x0008DB38
  68 65 6C 6C 6F 20 77 6F 72 6C 64                                  hello world
Length = 0xB
ByteOffset = NULL
Key = NULL
```

Na výpisu po volání funkce **NtReadFile** a při volání **NtWriteFile** je vidět formát, ve kterém jsou vypisovány buffery. Kvůli formátování této práce byly výpisy upraveny z původních 32 bytů na řádek na 16 bytů na řádek. Výpisy bufferů jsou u obecných bufferů jak hexadecimální, tak se znakovým náhledem, ve kterém jsou netisknutelné znaky zobrazeny jako tečky. U bufferů číselných je výpis pouze hexadecimální. Lze si zde také všimnout, že velikost bufferu je dána nějakým dalším argumentem funkce. U známých závislostí, jako například u funkcí **NtReadFile** a **NtWriteFile**, jsou tyto závislosti brány v potaz a výpisy tak vystihují přesná data, která volání použilo.

5 Srovnání s podobným software

V této části srovnáme prezentovanou implementaci řešení s již existujícím software, který se v nějaké formě zabývá sledováním procesů za běhu na systémech s operačním systémem Windows s jádrem NT. Srovnáváme pouze funkcionalitu produktů, nikoliv uživatelské rozhraní či jiné, pro tuto práci nevýznamné, aspekty. Navíc se omezujeme pouze na modul SystemSpy, který je hlavním výsledkem této práce.

Ve srovnání s podobnými produkty nabízí modul SystemSpy jednoznačně největší úroveň informací, které dokáže o daném volání získat. Oproti některým přímočarým řešením zaostává naše implementace svým výkonem, který je srovnatelný s těmi produkty, které implementují techniky user mode hákování. Příčiny většího zpomalení při hákování více procesů zároveň byly rozebrány v posledním odstavci části 3.2 této práce.

5.1 Process Monitor v1.12

URL: http://www.microsoft.com/technet/sysinternals/utilities/processmonitor.mspx

Techniky: Použití vrstevnatého modelu ovladačů a filtrace, Hákování System Service Descriptor Table

Process Monitor je nástroj pro sledování práce se soubory na disku a se systémovým registrem. Tento nástroj je naprogramován tak, aby efektivně poskytoval základní informace o aktivitách jako jsou vytváření, mazání, otevírání, zavírání a získávání a nastavování informací o souborech a adresářích na disku a o podobných operacích nad klíči a hodnotami systémového registru. Dále lze tento nástroj použít pro sledování vzniku a zániku procesů a vláken. Pro sledování operací se soubory používá filtrovací ovladač, zatímco pro sledování operací s registry byla použita technika hákování vybraných funkcí v SSDT. Nástroj sleduje všechny procesy v systému a případně filtruje volání od procesů, které uživatel nechce sledovat.

Největší výhodou tohoto nástroje je rychlost. Narozdíl od *Software* jsou handlery hákovaných funkcí implementovány v samotném ovladači, tudíž se nevykonává žádná práce navíc a aplikační volání je odbaveno prakticky ihned.

Mezi nevýhody ve srovnání s modulem SystemSpy patří malá detailnost zobrazovaných informací. Uživateli jsou poskytnuta jen základní data jako např. jméno otevíraného souboru, délka a offset zapisovaných dat, u registrových hodnot i zapisovaná data, návratová hodnota volání funkce. Chybí možnost detailního náhledu na čtená a zapisovaná data u souborů a na další parametry volaných operací. Navíc je tento nástroj určen pouze pro sledování operací se soubory a registrem, jeho pokrytí aktivit sledovaných procesů je tak značně menší než u *Software*.

5.2 API Monitor 1.1.1.70

URL: http://www.apimonitor.com/

Techniky: Hákování knihovních funkcí v user mode

API Monitor je komerční řešení pro sledování volání API funkcí v user mode. Na adresu vybraných funkcí v paměti sledovaného procesu umístí instrukci debug breakpoint a chová se tak jako debugger dané aplikace.

Hlavní výhodou tohoto produktu je rozsah pokrytí aktivit procesu. Výrobce deklaruje znalost až 4000 volání ve více jak 80 knihovnách. Modul SystemSpy rozpozná přibližně 300 volání. Další příjemná vlastnost je existence přednastavených skupin funkcí. Uživatel se tak nemusí obávat, že opomene zadat sledování nějaké funkce, pokud má zajem o jednu z přednastavených kategorií funkcí.

Mezi nevýhody patří vysoká režie sledování. Sledovaný proces je velmi výrazně zpomalen. Další nevýhodou je nemožnost sledovat více procesů najednou. Přestože úroveň informací je větší než u Process Monitoru, ve srovnání s modulem SystemSpy zaostává. API Monitor zná jen základní typy argumentů a pro většinu volání tak není schopen zobrazit relevantní informace. Nevýhodou je i samotná technika sledování, která může být sledovaným procesem jednoduše obcházena.

5.3 API Monitor 1.5b

URL: http://www.rohitab.com/apimonitor/

Techniky: Hákování knihovních funkcí v user mode

Název API Monitor sdílí i další produkt pro sledování volání API. Na rozdíl od svého jmenovce používá pro sledování změnu ukazatele v IAT.

API Monitor nabízí poměrně širokou škálu sledovaných funkcí, které jsou rozděleny do skupin podle jejich funkce. Oproti prvnímu API Monitoru umí také sledovat více procesů najednou a nemá takové výkonostní dopady na sledované procesy.

Nevýhodou tohoto software je použitá technika hákování IAT. Tato technika je ještě méně spolehlivá než přímá náhrada instrukcí hákované funkce v user mode. Úroveň poskytovaných informací je ještě o něco menší než v případě prvního API Monitoru.

5.4 Auto Debug Professional V4.0

URL: http://www.autodebug.com/

Techniky: Hákování knihovních funkcí v user mode

Další z komerčních řešení je rodina produktů Auto Debug. Systém sledování je založen na stejném principu jako prvně jmenovaný API Monitor, tedy využití instrukce debug breakpoint pro zachycení požadovaných volání. Stejně jako oba API Monitory slouží tento produkt ke sledování volání API funkcí, které exportují user mode knihovny.

Oproti předchozím produktům je zde i možnost detailně sledovat knihovny, ke kterým existuje debugovací informace ve formátu .pdb. Výhodou tohoto produktu oproti modulu SystemSpy je opět rozsah pokrytí.

Nevýhodou je zde samotná technika, kterou lze obejít z pozice sledovaného procesu. Detail zobrazovaných informací je zde největší ze všech zmíněných produktů. Pokud je k dispozici debugovací informace sledovaného procesu je možné sledovat i interní třídy a funkce

sledovaného procesu. V obecném případě je však úroveň informací stále mnohem menší než nabízí modul SystemSpy.

5.5 Strace for NT 0.3

URL: http://www.bindview.com/Services/RAZOR/Utilities/Windows/strace_readme.cfm

Techniky: Hákování System Service Descriptor Table

Strace je jednoduchý nástroj pro sledování systémových služeb. Skládá se z klientské aplikace a ovladače, který provádí hákovaných funkcí v SSDT a SSDT Shadow.

Jednoznačnou výhodou tohoto nástroje je jednoduchost a rychlost použití, ale také minimální dopad na výkon sledovaných procesů. Oproti *Software* umí tento nástroj navíc sledovat funkce grafického rozhraní.

Nevýhoda je v pouze základní nabídce informací o voláních. Kromě řetězců, několika základních struktur a konstant dokáže Strace zobrazit jen číselné hodnoty argumentů sledovaných funkcí.

6 Závěr

Výsledkem této práce je poměrně solidní základ systému pro monitorování a ovlivňování chování procesů. Ačkoliv je *Software* v prezentované verzi použitelný pro cíle definované na začátku práce, je zde především vysoký potenciál pro budoucí vývoj.

Návrh vychází z jedné možné techniky, hákování v SSDT, která však má svoje omezení. V části 2.2 této práce jsou však zmíněny další techniky, které jsme pro splnění definovaných cílů označili jako méně vhodné. Přesto tyto techniky nabízejí mnoho možností, které by mohli přispět k zlepšení *Software*. Do budoucna se tak nabízí eventualita implementace více než jedné techniky, mezi kterými by si uživatel systému vybíral na základě svých aktuálních potřeb nebo je dokonce kombinoval. Například hákování funkcí v user mode a v SSDT Shadow by mohlo výrazně rozšířit rozsah pokrytí aktivit sledovaných procesů.

Abychom mohli uvažovat o nasazení výsledného řešení pro použití širším okruhem uživatelů, je nezbytné implementovat lepší ovládací prvky jak celého systému, tak i jednotlivých modulů. Dále je třeba nabídnout standardní instalační proceduru a také možnost používání *Software* s omezenými právy uživatele. Napomoci může implementace grafického interface s moderním designem a intuitivním ovládáním. Mohli bychom uvažovat i o sledování procesů přes síť tak, že na cílovém počítači by bylo instalováno pouze jádro sledovacího systému a z okolních počítačů by se klient uživatelského rozhraní mohl připojit a sledovat aktivity vybraných procesů. Dále je možné implementovat podporu sledování již při startu systému apod.

Dalším základním prvkem v budoucím vývoji je tvorba dalších užitečných modulů. V tomto ohledu by mohlo výrazně pomoci zveřejnění API pro externí moduly tak, aby nezávislí programátoři mohli přispívat svými moduly. K tomu by mohlo přispět i napsání hlavičkových souborů pro další kompilátory jako je např. kompilátor Microsoft Visual Studia, ale i podpora jiných programovacích jazyků. Dále sem patří i další rozvoj modulu SystemSpy, který by v budoucnu mohl nabídnout velmi sofistikované funkce pro analýzu chování procesů s možností interaktivně zasahovat do vykonávání kódu sledovaných procesů. Nabízí se také možnosti definovat skupiny pravidel pro sledování určité kategorie aktivit procesu, sledování vzájemných interakcí dvou a více procesů, přesměrování některých aktivit do virtuálního prostředí tak, aby nedocházelo ke změnám v hostitelském operačním systému atd. Na základě modulu SystemSpy by také mohla vzniknout práce srovnávající chování konkrétních exemplářů různých druhů software s důrazem na analýzu moderního malware.

Prakticky nezbytnou součástí dalšího vývoje je práce na zlepšení výkonu celého systému. V současné verzi je v případě zavedení system-wide hook velmi citelně zasažen výkon operačního systému. Úzkým hrdlem v návrhu je existence pouze jednoho bufferu pro komunikaci mezi službou a ovladačem v případě zachycení volání hákované funkce. Při zvětšení počtu těchto bufferů tak lze počítat i s několikanásobným zvýšením výkonu při hákování více procesů najednou. V ideálním případě by mělo být možné nasadit system-wide hook tak, aby uživatel při běžné práci nepoznal rozdíl v odezvě běžících aplikací.

Zdokonalení lze dosáhnout i rozšířením API pro externí moduly. Užitečné by například mohly být funkce, které by umožňovaly přímo z modulu dynamicky přidávat a odebírat hákovací i filtrovací pravidla.

Tato práce vznikla v době, kdy se na světlo světa dostala nová verze operačního systému Windows nazvaná Windows Vista. V předkládané podobě výsledky této práce nejsou kompatibilní s tímto systémem, ale právě podpora tohoto systému by byla velkým přínosem pro možnost rozšíření tohoto díla do budoucna. Tato práce také není kompatibilní se stále populárnější architekturou 64 bitových procesorů a stejně jako u případu Windows Vista je rozšíření pro tuto platformu jeden z důležitých cílů v dalším vývoji.

Byl bych velmi rád a věřím, že se najde dost sil k tomu, aby na základě mé práce vzniknul užitečný a použitelný balík produktů, který osloví široký okruh uživatelů i programátorů. Tímto bych rád poprosil čtenáře mé práce, aby se mi v případě zájmu podílet se na dalším rozvoji výsledků této práce ozvali, popř. aby informovali studenty ve svém okolí, kteří by takový zájem mohli mít.

Příloha A – API externích modulů

Použití funkcí API externích modulů a příslušných struktur vyžaduje použití hlavičkového souboru `<core-exports/wcpapi.h>`. Jednotlivé moduly jsou standardní DLL knihovny s exportovanými funkcemi handlerů, které mohou volitelně implementovat **DllMain** v případně nutnosti vlastní inicializace a finalizace.

Základní řízení

HookHandler

HookHandler je aplikačně definovaná funkce, která slouží jako vstupní bod obsluhy hákované funkce.

```
WCP_ERR WCPAPI HookHandler(
  WCP_SESSION session
);
```

Parametry

session
[in] Identifikátor relace, který je většinou vyžadován při volání ostatních API.

Návratová hodnota

Návratová hodnota závisí na funkci handleru, tato hodnota je pak vnímána handlerem vyšší úrovně jako návratová hodnota z volání **WcpCallNextHandler**, případně jako návratová hodnota samotné hákované funkce.

Pokud handler nechce změnit chování hákovaného procesu, měl by dál předat hodnotu, kterou získá z volání **WcpCallNextHandler**.

Poznámky

Jelikož mají všechny handlery pro hákované funkce jednotný předpis, je možné použít jednu funkci handleru pro obsluhu více hákovaní. Je však možno definovat i více handlerů v jednom modulu. Každý handler ale musí být linkován jako exportovaná funkce. Přiřazení hákování k handlerům se děje v souboru pravidel.

Příklad

HookHandler nutně používájí všechny moduly, bývají však definovány s různými názvy. Například v modulu `modules\samples\minimal` je jediný hook handler **PassThrough**, zatímco v modulu `modules\samples\fileredir` jsou hook handlery funkce **CreateOpen**, **Close** a **Delete**.

Související informace

WcpCallNextHandler

WcpCallNextHandler

Funkce **WcpCallNextHandler** zavolá další funkci v hook chain, tedy obsluhu hákované funkce nižší úrovně.

```
WCP_ERR WcpCallNextHandler(
  WCP_SESSION session,
  NTSTATUS *status
);
```

Parametry

session

[in] Identifikátor relace, který byl předán handleru.

status

[out] Návratová hodnota obsluhy nižší úrovně.

Návratová hodnota

Pokud funkce uspěje, návratová hodnota je WCP_ERR_SUCCESS.

Pokud se systém hákování ukončuje, např. v důsledku vypínání počítače, je návratová hodnota WCP_ERR_SHUTDOWN_DETECTED.

Pokud bylo hákované vlákno ukončeno dříve než všechny nižší handlery dokončily svoji práci, je návratová hodnota WCP_ERR_THREAD_TERMINATED.

Pro testování úspěšnosti volání lze použít předdefinovaná makra WCP_ERR_SUCCEED a WCP_ERR_FAILED.

Poznámky

Pokud **WcpCallNextHandler** skončí úspěšně a není požadována modifikace chování, vrací handler hodnotu získanou v parametru *status*, přetypovanou na WCP_ERR. Pokud **WcpCallNextHandler** skončí s chybou, není *status* naplněn, jeho hodnota je nedefinovaná a vrácená chyba by se měla ihned propagovat vyššímu handleru.

Zavoláním funkce **WcpCallNextHandler** předá funkce řízení handleru na nižší úrovni v hook chain. Pokud žádný takový nižší handler neexistuje, je kontaktován ovladač systému, který zařídí volání aktuální obsluhy hákované funkce.

Příklad

Funkci **WcpCallNextHandler** nutně používájí všechny moduly, správné použití této funkce tak lze nalézt i v modulu `modules\samples\minimal`.

Související informace

HookHandler, **WcpChangeArgumentList**

Informace o hákování a jeho kontextu

WcpGetCallerInfo

Funkce **WcpGetCallerInfo** slouží k získání informací o hákovaném procesu a vláknu.

```
void WcpGetCallerInfo(
  WCP_SESSION session,
  PWCP_PROCESS_INFORMATION info
);
```

Parametry

session
[in] Identifikátor relace, který byl předán handleru.

info
[out] Ukazatel na strukturu **WCP_PROCESS_INFORMATION**, která bude naplněna informacemi o hákovaném procesu a vláknu.

Návratová hodnota

Tato funkce nemá návratovou hodnotu, vždy uspěje.

Poznámky

Struktura, na níž ukazuje parametr *info*, obsahuje statická i dynamická data. Dynamická data ve struktuře mají garantovanou platnost pouze do volání libovolné funkce, jejíž návratová hodnota je WCP_ERR. Pokud chce volající použít data získaná pomocí **WcpGetCallerInfo** i později, musí si udělat jejich kopii do lokální paměti.

Příklad

Funkci **WcpGetCallerInfo** používají například moduly `modules\samples\counter2` a `modules\samples\memory`.

Související informace

WCP_PROCESS_INFORMATION, **WcpGetFunctionName**, **WcpGetFunctionArgs**

WcpGetFunctionName

Funkce **WcpGetFunctionName** slouží k získání jména hákované funkce.

```
void WcpGetFunctionName(
  WCP_SESSION session,
  char **name
);
```

Parametry

session
[in] Identifikátor relace, který byl předán handleru.

name
[out] Ukazatel na proměnnou, která bude naplněna ukazatelem na jméno hákované funkce.

Návratová hodnota
Tato funkce nemá návratovou hodnotu, vždy uspěje.

Poznámky
Ukazatel na jméno hákované funkce má garantovanou platnost pouze do volání libovolné funkce, jejíž návratová hodnota je WCP_ERR. Pokud chce volající použít jméno funkce získané pomocí **WcpGetFunctionName** i později, musí si udělat jejich kopii do lokální paměti.

Příklad
Funkci **WcpGetFunctionName** používá například modul `modules\samples\counter` pro rozlišení hákování pocházejících z volání **NtCreateFile** a **NtOpenFile**.

Související informace
WcpGetCallerInfo, **WcpGetFunctionArgs**

WcpGetFunctionArgs

Funkce **WcpGetFunctionArgs** slouží k získání pole argumentů hákované funkce a jejich počtu.

```
void WcpGetFunctionArgs(
  WCP_SESSION session,
  PAPI_ARG *argv,
  int *argc
);
```

Parametry
session
[in] Identifikátor relace, který byl předán handleru.

argv
[out] Ukazatel na proměnnou, která bude naplněna ukazatelem na pole argumentů hákované funkce. Toto pole má v bytech velikost sizeof(API_ARG)**argc*. Tento argument může být NULL, pokud ukazatel na pole argumentů není vyžadován.

argc
[out] Ukazatel na proměnnou, která bude naplněna počtem argumentů hákované funkce. Tento parametr může být NULL, pokud počet argumentů není vyžadován.

Návratová hodnota

Tato funkce nemá návratovou hodnotu, vždy uspěje.

Poznámky

Ukazatel na pole argumentů hákované funkce má garantovanou platnost pouze do volání libovolné funkce, jejíž návratová hodnota je WCP_ERR. Pokud chce volající použít tato data i později, musí si udělat jejich kopii do lokální paměti.

Příklad

Funkci **WcpGetFunctionArgs** používá například modul `modules\samples\memory`.

Související informace

WcpGetCallerInfo, **WcpGetFunctionName**

Práce s pamětí

WcpReadCallerMemory

Funkce **WcpReadCallerMemory** slouží ke čtení úseku paměti z adresového prostoru hákovaného procesu.

```
WCP_ERR WcpReadCallerMemory(
  WCP_SESSION session,
  REMOTE_POINTER address,
  PVOID buffer,
  ULONG size
);
```

Parametry

session

[in] Identifikátor relace, který byl předán handleru.

address

[in] Ukazatel v adresovém prostoru hákovaného procesu, odkud se bude číst.

buffer

[out] Ukazatel do lokální zapisovatelné paměti velikosti alespoň *size* bytů, kam budou uložena přečtená data z hákovaného procesu.

size

[in] Počet bytů, které se mají přečíst v paměti hákovaného procesu z adresy *address* a zároveň velikost lokální zapisovatelné paměti, na kterou ukazuje *buffer*.

Návratová hodnota

Pokud funkce uspěje, návratová hodnota je WCP_ERR_SUCCESS.

Pokud je ukazatel v paměti hákovaného procesu *address* neplatný, nebo odkazuje

na paměť, která není čitelná v rozsahu alespoň *size* bytů, pak je návratová hodnota WCP_ERR_ACCESS_VIOLATION a obsah lokální paměti na adrese *buffer* není definován.

Pokud se systém hákování ukončuje, např. v důsledku vypínání počítače, je návratová hodnota WCP_ERR_SHUTDOWN_DETECTED.

Pokud bylo hákované vlákno ukončeno dříve než všechny nižší handlery dokončily svoji práci, je návratová hodnota WCP_ERR_THREAD_TERMINATED.

Pro testování úspěšnosti volání lze použít předdefinovaná makra WCP_ERR_SUCCEED a WCP_ERR_FAILED.

Poznámky

Pokud **WcpReadCallerMemory** neskončí úspěšně a vrácená chyba není WCP_ERR_ACCESS_VIOLATION, volající handler by neměl dále pokračovat v obsluze a vrácená chyba by se měla ihned propagovat vyššímu handleru.

Při práci s pamětí v hákovaném procesu je třeba mít na paměti, že všechna přečtená data jsou lokální v rámci cizího procesu. Přečtené hodnoty ukazatelů jsou platné pouze v adresovém prostoru cizího procesu a při práci se složitějšími strukturami může být nezbytné několikanásobné čtení pro získání všech informací.

Příklad

Funkci **WcpReadCallerMemory** používá modul `modules\samples\memory`. Složitější práci s pamětí v hákovaném procesu, včetně výše zmíněného několikanásobného čtení, používá modul `modules\samples\fileredir`.

Související informace

WcpWriteCallerMemory

WcpWriteCallerMemory

Funkce **WcpWriteCallerMemory** slouží k zapsání dat do paměti v adresovém prostoru hákovaného procesu.

```
WCP_ERR WcpWriteCallerMemory(
  WCP_SESSION session,
  REMOTE_POINTER address,
  PVOID buffer,
  ULONG size
);
```

Parametry

session

[in] Identifikátor relace, který byl předán handleru.

address

[in] Ukazatel v adresovém prostoru hákovaného procesu, kam se bude zapisovat.

buffer

[in] Ukazatel na data v lokální paměti velikosti *size* bytů.

size

[in] Počet bytů, které se mají zapsat do paměti hákovaného procesu na adresu *address* a zároveň minimální velikost lokální paměti, na kterou ukazuje *buffer*.

Návratová hodnota

Pokud funkce uspěje, návratová hodnota je WCP_ERR_SUCCESS.

Pokud je ukazatel v paměti hákovaného procesu *address* neplatný, nebo odkazuje na paměť, která není zapisovatelná v rozsahu alespoň *size* bytů, pak je návratová hodnota WCP_ERR_ACCESS_VIOLATION.

Pokud se systém hákování ukončuje, např. v důsledku vypínání počítače, je návratová hodnota WCP_ERR_SHUTDOWN_DETECTED.

Pokud bylo hákované vlákno ukončeno dříve než všechny nižší handlery dokončily svoji práci, je návratová hodnota WCP_ERR_THREAD_TERMINATED.

Pro testování úspěšnosti volání lze použít předdefinovaná makra WCP_ERR_SUCCEED a WCP_ERR_FAILED.

Poznámky

Pokud **WcpWriteCallerMemory** neskončí úspěšně a vrácená chyba není WCP_ERR_ACCESS_VIOLATION, volající handler by neměl dále pokračovat v obsluze a vrácená chyba by se měla ihned propagovat vyššímu handleru.

Při práci s pamětí v hákovaném procesu je třeba mít na paměti, že všechna zapisovaná data jsou lokální v rámci cizího procesu. Pokud například dochází k zapisování hodnot typu ukazatel, musí tyto ukazatelé být platné v rámci cizího procesu. Zapisovat lze pouze do paměti, která existuje v cizím procesu a je označena jako zapisovatelná. Toto lze zaručit například alokací pomocí volání **WcpCallerAlloc**. Lze však i přímo měnit zapisovatelné části paměti cizího procesu, jako je například zásobník nebo pamět určená k naplnění příslušným voláním. Ve všech případech je nezbytné dbát na integritu dat, jinak může dojít k pádu hákovaného procesu.

Příklad

Funkci **WcpWriteCallerMemory** používá modul `modules\samples\fileredir`, který názorně řeší úskalí zapisování složitějších struktur v hákovaném procesu.

Související informace

WcpReadCallerMemory, **WcpCallerAlloc**, **WcpCallerFree**

WcpCallerAlloc

Funkce **WcpCallerAlloc** alokuje nebo získá referenci na pojmenovanou paměť v hákovaném procesu.

```
WCP_ERR WcpCallerAlloc(
```

```
  WCP_SESSION session,
  ULONG tag,
  PREMOTE_POINTER ptr,
  PULONG size,
  PULONG protect
);
```

Parametry

session
[in] Identifikátor relace, který byl předán handleru.

tag
[in] Číselné pojmenování paměti.

ptr
[in,out] Ukazatel na proměnnou, která při volání obsahuje hodnotu ukazatele v paměti hákovaného procesu na adresu, kterou volající preferuje pro alokování. Pokud je nastavena na NULL, systém zvolí libovolnou adresu pro alokování. Na výstupu se vrací skutečná adresa alokované paměti v hákovaném procesu.

size
[in, out] Ukazatel na proměnnou, která na vstupu určuje velikost alokované paměti. Na výstupu se vrací skutečná velikost alokované paměti v hákovaném procesu.

protect
[in, out] Ukazatel na proměnnou, která na vstupu určuje typ ochrany paměti. Na výstupu se vrací skutečný typ ochrany paměti.

Návratová hodnota

Pokud funkce uspěje, a nová pamět byla v hákovaném procesu alokována, je návratová hodnota WCP_ERR_SUCCESS. Pokud funkce uspěje tak, že nalezne existující záznam s daným *tag*em, tedy nová paměť nebyla alokována, je návratová hodnota WCP_ERR_INF_NOT_CREATED.

Pokud je lokální seznam segmentů plný, funkce neuspěje a návratová hodnota je WCP_ERR_STRUCTURE_FULL. Pro více informací viz poznámky.

Pokud se systém hákování ukončuje, např. v důsledku vypínání počítače, je návratová hodnota WCP_ERR_SHUTDOWN_DETECTED.

Pokud bylo hákované vlákno ukončeno dříve než všechny nižší handlery dokončily svoji práci, je návratová hodnota WCP_ERR_THREAD_TERMINATED.

Pro testování úspěšnosti volání lze použít předdefinovaná makra WCP_ERR_SUCCEED a WCP_ERR_FAILED.

Poznámky

Systém alokace pomocí **WcpCallerAlloc** zajišťuje programátorům handlerů pohodlnou práci s pamětí v hákovaných procesech. Typicky je třeba alokovat paměť ve všech hákovaných procesech. Aby programátor handleru nemusel řešit správu procesů pro každý takový proces, jednoduše pojmenuje paměť, kterou chce využít unikátním *tag*em v rámci modulu.

Volání **WcpCallerAlloc** se nejprve pokusí vyhledat v lokálním seznamu alokovaných

segmentů paměti záznam pro aktuálně hákovaný proces, který má příslušný *tag* a náleží volajícímu modulu. Pokud takový záznam existuje, jsou vstupní hodnoty parametrů *ptr*, *size* a *protect* ignorovány a jsou pouze naplněny informacemi o již existujícím paměťovém segmentu.

Pokud záznam pro hákovaný proces, daný *tag* a volající modul neexistuje, je kontaktován ovladač, který zařídí alokaci nové paměti v hákovaném procesu. V případě úspěchu se informace uloží do lokálního seznamu.

V handleru se tak vždy použije volání **WcpCallerAlloc** a není třeba si pamatovat pro každý proces, zda je v něm daná paměť alokována či nikoliv. Pro programátora handleru je situace transparentní tak, jakoby pracoval vždy jen s jedním procesem.

Lokální seznam alokováné paměti je omezen na 256 položek na jeden proces. Toto omezení se týká procesu a tedy se jedná o maximální počet položek všech modulů, které s hákovaným procesem pracují. Namísto alokace více menších segmentů je proto lepší alokovat jeden větší segment a logicky jej rozdělit.

Pro uvolnění paměti alokované pomocí **WcpCallerAlloc** slouží volání **WcpCallerFree**.

Parametr *protect* je na výstupu naplněn jen tehdy, když je nalezen existující záznam, nikoliv pokud se alokuje nová paměť. V takovém případě, pokud se alokace zdaří, má nově alokovaná paměť vždy požadovanou ochranu.

Hodnoty parametru protect jsou totožné jako při volání Windows API **VirtualAlloc**.

Výstupní hodnota v parametru *ptr* se typicky používá do volání **WcpWriteCallerMemory** a případně **WcpReadCallerMemory**. Pokud je účelem práce s pamětí dostat do hákovaného procesu neměnná data, lze data do procesu zapsat pouze jednou v případě, že návratová hodnota při úspěšném volání **WcpCallerAlloc** byla WCP_ERR_SUCCESS. V druhém případě úspěšného volání, tedy při návratu WCP_ERR_INF_NOT_CREATED, již není třeba data znovu zapisovat.

Pokud návratová hodnota funkce je WCP_ERR_SHUTDOWN_DETECTED nebo WCP_ERR_THREAD_TERMINATED, neměl by volající handler dále pokračovat v obsluze a vrácená chyba by se měla ihned propagovat vyššímu handleru.

Příklad

Funkci **WcpCallerAlloc** používá modul `modules\samples\fileredir`.

Související informace

WcpCallerFree, **WcpReadCallerMemory**, **WcpWriteCallerMemory**

WcpCallerFree

Funkce **WcpCallerFree** uvolňuje paměť alokovanou v hákovaném procesu pomocí **WcpCallerAlloc**.

```
WCP_ERR WcpCallerFree(
  WCP_SESSION session,
  ULONG tag
);
```

Parametry

session
[in] Identifikátor relace, který byl předán handleru.

tag
[in] Číselné pojmenování paměti.

Návratová hodnota

Pokud funkce uspěje, návratová hodnota je WCP_ERR_SUCCESS.

Pokud funkce neuspěje, protože danému *tag*u neodpovídá v aktuálně hákovaném procesu žádná alokováná paměť, je návratová hodnota WCP_ERR_ITEM_NOT_FOUND.

Pokud se systém hákování ukončuje, např. v důsledku vypínání počítače, je návratová hodnota WCP_ERR_SHUTDOWN_DETECTED.

Pokud bylo hákované vlákno ukončeno dříve než všechny nižší handlery dokončily svoji práci, je návratová hodnota WCP_ERR_THREAD_TERMINATED.

Pro testování úspěšnosti volání lze použít předdefinovaná makra WCP_ERR_SUCCEED a WCP_ERR_FAILED.

Poznámky

Pokud návratová hodnota funkce je WCP_ERR_SHUTDOWN_DETECTED nebo WCP_ERR_THREAD_TERMINATED, neměl by volající handler dále pokračovat v obsluze a vrácená chyba by se měla ihned propagovat vyššímu handleru.

Příklad

Funkci **WcpCallerFree** používá modul `modules\samples\fileredir`.

Související informace

WcpCallerAlloc

Modifikace chování

WcpChangeArgumentList

Funkce **WcpChangeArgumentList** slouží ke změně chování hákovaného procesu pomocí změny argumentů hákované funkce.

```
void WcpChangeArgumentList(
  WCP_SESSION session,
  PAPI_ARG arg_list
);
```

Parametry

session

[in] Identifikátor relace, který byl předán handleru.

arg_list
[in] Ukazatel na nové pole argumentů.

Návratová hodnota

Tato funkce nemá návratovou hodnotu, vždy uspěje.

Poznámky

Pole argumentů, na které ukazuje *arg_list*, musí obsahovat minimálně tolik argumentů, kolik jich přijímá hákovaná funkce. Počet argumentů hákované funkce lze získat pomocí volání **WcpGetFunctionArgs**.

Změna argumentů se projeví ve všech nižších handlerech při volání **WcpCallNextHandler**, ale také ve všech vyšších handlerech po návratu z aktuálního handleru. Nižší handlery navíc nepoznají, že vůbec došlo k nějaké modifikaci argumentů.

Příklad

Funkci **WcpChangeArgumentList** používá modul `modules\samples\fileredir`.

Související informace

WcpGetFunctionArgs, **WcpCallNextHandler**

Správa událostí

WcpRegisterProcessNotificationRoutine

Funkce **WcpRegisterProcessNotificationRoutine** slouží k zaregistrování funkce, která získá řízení vždy, když dojde ke vzniku nebo zániku procesu v systému.

```
WCP_ERR WcpRegisterProcessNotificationRoutine(
  WCP_NOTIFICATION_PROCESS_ROUTINE routine
);
```

Parametry

routine
[in] Adresa funkce, která získá řízení při vzniku nebo zániku procesu v systému.

Návratová hodnota

Pokud funkce uspěje, návratová hodnota je WCP_ERR_SUCCESS.

Pokud funkce neuspěje z důvodu nedostatku paměti, je návratová hodnota WCP_ERR_NOT_ENOUGH_MEMORY.

Pokud již daná funkce je registrována pro notifikace procesů nebo vláken, pak volání skončí s návratovou hodnotou WCP_ERR_ALREADY_EXISTS.

Poznámky

Registraci notifikační funkce je vhodné umístit do **DllMain** modulu.

Pokud modul registruje nějakou notifikační funkci, je nezbytné ji odregistrovat pomocí **WcpUnregisterNotificationRoutine** před odebráním modulu. Nejsnáze to lze udělat voláním této funkce v **DllMain**, který je systémem zavolán před odebráním modulu.

Příklad

Modul `modules\samples\notifications` demonstruje použití notifikačních funkcí.

Související informace

WCP_NOTIFICATION_PROCESS_ROUTINE, **WcpUnregisterNotificationRoutine**, **WcpRegisterThreadNotificationRoutine**

WcpRegisterThreadNotificationRoutine

Funkce **WcpRegisterThreadNotificationRoutine** slouží k zaregistrování funkce, která získá řízení vždy, když dojde ke vzniku nebo zániku vlákna v systému.

```
WCP_ERR WcpRegisterThreadNotificationRoutine(
  WCP_NOTIFICATION_THREAD_ROUTINE routine
);
```

Parametry

routine

[in] Adresa funkce, která získá řízení při vzniku nebo zániku vlákna v systému.

Návratová hodnota

Pokud funkce uspěje, návratová hodnota je WCP_ERR_SUCCESS.

Pokud funkce neuspěje z důvodu nedostatku paměti, je návratová hodnota WCP_ERR_NOT_ENOUGH_MEMORY.

Pokud již daná funkce je registrována pro notifikace procesů nebo vláken, pak volání skončí s návratovou hodnotou WCP_ERR_ALREADY_EXISTS.

Poznámky

Registraci notifikační funkce je vhodné umístit do **DllMain** modulu.

Pokud modul registruje nějakou notifikační funkci, je nezbytné ji odregistrovat pomocí **WcpUnregisterNotificationRoutine** před odebráním modulu. Nejsnáze to lze udělat voláním této funkce v **DllMain**, který je systémem zavolán před odebráním modulu.

Příklad

Modul `modules\samples\notifications` demonstruje použití notifikačních funkcí.

Související informace

WCP_NOTIFICATION_THREAD_ROUTINE, **WcpUnregisterNotificationRoutine**, **WcpRegisterProcessNotificationRoutine**

WcpUnregisterNotificationRoutine

Funkce **WcpUnregisterNotificationRoutine** slouží k odregistrování notifikační funkce, která byla registrována pomocí **WcpRegisterProcessNotificationRoutine** nebo pomocí **WcpRegisterThreadNotificationRoutine**.

```
WCP_ERR WcpUnregisterNotificationRoutine(
  WCP_NOTIFICATION_ROUTINE routine
);
```

Parametry

routine
[in] Adresa funkce, která má být odregistrována.

Návratová hodnota

Pokud funkce uspěje, návratová hodnota je WCP_ERR_SUCCESS.

Pokud funkce nebyla registrována pro notifikace, volání skončí s návratovou hodnotou WCP_ERR_ITEM_NOT_FOUND.

Poznámky

Pokud modul registruje nějakou notifikační funkci, je nezbytné ji odregistrovat pomocí **WcpUnregisterNotificationRoutine** před odebráním modulu. Nejsnáze to lze udělat voláním této funkce v **DllMain**, který je systémem zavolán před odebráním modulu.

Příklad

Modul `modules\samples\notifications` demonstruje použití notifikačních funkcí.

Související informace

WcpRegisterProcessNotificationRoutine, **WcpRegisterThreadNotificationRoutine**

Použité struktury a prototypy funkcí

WCP_PROCESS_INFORMATION

Tato struktura obsahuje informace, které může handler získat o hákovaném procesu a vláknu. Strukturu naplňuje volání **WcpGetCallerInfo**.

```
typedef struct WCP_CALLER_INFORMATION
{
```

```
  ULONG pid;
  ULONG tid;
  wchar_t process_path[MAX_PATH];
  wchar_t *process_name;
} WCP_PROCESS_INFORMATION,*PWCP_PROCESS_INFORMATION;
```

Položky

pid
Systémový identifikátor hákovaného procesu.

tid
Systémový identifikátor hákovaného vlákna.

process_path
Úplná cesta k hlavnímu modulu hákovaného procesu v systémovém formátu, např. `\Device\HarddiskVolume1\Program Files\Internet Explorer\iexplore.exe`.

process_name
Ukazatel na jméno hlavního modulu procesu bez cesty, např. `iexplore.exe`. Typicky se jedná o ukazatel do *process_path*.

Poznámky

Pro informace ohledně platnosti dat v této struktuře viz **WcpGetCallerInfo**.

Související informace

WcpGetCallerInfo

WCP_NOTIFICATION_PROCESS_ROUTINE

Prototyp funkce pro notifikace při vzniku a zániku procesů, která se registruje pomocí **WcpRegisterProcessNotificationRoutine**.

```
void WCPAPI (*WCP_NOTIFICATION_PROCESS_ROUTINE) (
  int create,
  ULONG pid,
  ULONG ppid,
  wchar_t *name
);
```

Položky

create
TRUE, pokud proces vzniká, FALSE, pokud zaniká.

pid
Systémový identifikátor procesu.

ppid
Systémový identifikátor předka procesu.

name

Úplná cesta k hlavnímu modulu hákovaného procesu v systémovém formátu, např. `\Device\HarddiskVolume1\Program Files\Internet Explorer\iexplore.exe`.

Poznámky

Parametry *ppid* a *name* jsou platné pouze pokud *create* je TRUE.

Související informace

WcpRegisterProcessNotificationRoutine

WCP_NOTIFICATION_THREAD_ROUTINE

Prototyp funkce pro notifikace při vzniku a zániku vláken, která se registruje pomocí **WcpRegisterThreadNotificationRoutine**.

```
void WCPAPI (*WCP_NOTIFICATION_THREAD_ROUTINE) (
  int create,
  ULONG tid,
  ULONG pid
);
```

Položky

create
TRUE, pokud vlákno vzniká, FALSE, pokud zaniká.

tid
Systémový identifikátor vlákna.

pid
Systémový identifikátor procesu, kterému vlákno náleží.

Poznámky

Parametr *pid* je platný pouze pokud *create* je TRUE.

Související informace

WcpRegisterThreadNotificationRoutine

Příloha B – Struktura přiloženého média

`\Bachelor thesis.pdf` – elektronická verze této práce

`\implementace` – kompilace *Software*, zdrojové kódy Software, další soubory

- `\setvenv.bat` – skript pro nastavení proměnné prostředí `WCPROOT` pro novou instanci shellu, ve které lze sestavit externí moduly, službu s klientem a všechny další nezbytné knihovny ke spuštění kromě ovladače
- `\build` – spustitelná kompilace *Software*
  - `\WcpSSDT.txt` – příklad výstupu modulu SystemSpy pro pravidla definovaná ve `wcp-rules-install.conf`
- `\core` – zrojové kódy služby, klienta a ovladače
  - `\Makefile` – makefile pro `mingw32-make` pro sestavení služby s klientem
  - `\driver` – zdrojové kódy ovladače
    - `\build` – sestavovací skripty pro ovladač
      - `\b.bat` sestavovací skript pro prostředí DDK Free Build Environment
      - `\bc.bat` sestavovací skript pro prostředí DDK Checked Build Environment
- `\core-exports` – hlavičkové soubory a knihovna s ModAPI pro tvorbu externích modulů
- `\modules` – zdrojové kódy externích modulů, každý adresář modulu obsahuje vlastní makefile pro sestavení modulu
  - `\samples` – zdrojové kódy jednoduchých demonstračních modulů
  - `\systemspy` – zdrojové kódy modulu SystemSpy pro sledování chování procesů
- `\modules-exports` – zkompilované moduly

Literatura

[1] Baker Art, Lozano Jerry: *The Windows 2000 Device Driver Book: A Guide for Programmers (2nd Edition)*

[2] Hoglund Greg, Butler Jamie: *Rootkits: Subverting the Windows Kernel*

[3] Nebbett Gary: *Windows NT/2000 Native API Reference*

[4] Oney Walter: *Programming the Microsoft Windows Driver Model, Second Edition*

[5] Russinovich Mark E., Solomon David A.: *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000*